

学习手册

目錄

SQLite学习手册

> 来源：SQLite学习手册

SQLite学习手册(目录)

在实际的应用中，SQLite作为目前最为流行的开源嵌入式关系型数据库，在系统的架构设计中正在扮演着越来越为重要的角色。和很多其它嵌入式NoSQL数据库不同的是，SQLite支持很多关系型数据库的基本特征，这在数据移植、程序演示等应用中有着不可替代的优势。从官方文档中我们可以获悉到，SQLite支持的数据量和运行效率都是非常骄人的，因此在海量数据的解决方案中，SQLite可以作为数据预计算的桥头堡，从而显著减少存储在关系型数据库服务器中的数据数量，最终提高系统的查询效率和运行期效率，同时也可以显著的降低数据备份的磁盘开销。这里提供了该系列博文的目录，以方便网友和我个人的学习与参阅。

Finally, if you are interseting in my series blogs, please pay more attention on my following ones, such as Redis, MongoDB, Lua and PostgreSQL. Thank you for your reading and comments, that will give me more effective encouragement and stimulate me to move ahead with stable and continuous.

SQLite学习手册(实例代码<一>)

http://www.cnblogs.com/stephen-liu74/archive/2012/02/07/2340780.html

一、获取表的Schema信息

二、常规数据插入

SQLite学习手册(实例代码<二>)

http://www.cnblogs.com/stephen-liu74/archive/2012/02/07/2341480.html

三、高效的批量数据插入

四、数据查询

如果您觉得这个系列的博客可以让您有所收获，请保持持续的关注。

如果您发现博客中有明显的纰漏，欢迎指正。

如果您有意进行技术上的交流，可以通过邮件共同探讨(stephenland74@hotmail.com)。

如果您有更好的建议或更好的实现方式，敬请赐教。

如果您已经是我的关注者，希望随后发布的Redis系列不会让您失望。

I am Stephen Liu.

SQLite学习手册(开篇)

一、简介：

SQLite是目前最流行的开源嵌入式数据库，和很多其他嵌入式存储引擎相比(NoSQL)，如BerkeleyDB、MemBASE等，SQLite可以很好的支持关系型数据库所具备的一些基本特征，如标准SQL语法、事务、数据表和索引等。事实上，尽管SQLite拥有诸多关系型数据库的基本特征，然而由于应用场景的不同，它们之间并没有更多的可比性。下面我们将列举一下SQLite的主要特征：

1). 管理简单，甚至可以认为无需管理。

2). 操作方便，SQLite生成的数据库文件可以在各个平台无缝移植。

3). 可以非常方便的以多种形式嵌入到其他应用程序中，如静态库、动态库等。

4). 易于维护。

综上所述，SQLite的主要优势在于灵巧、快速和可靠性高。SQLite的设计者们为了达到这一目标，在功能上作出了很多关键性的取舍，与此同时，也失去了一些对RDBMS关键性功能的支持，如高并发、细粒度访问控制(如行级锁)、丰富的内置函数、存储过程和复杂的SQL语句等。正是因为这些功能的牺牲才换来了简单，而简单又换来了高效性和高可靠性。

二、SQLite的主要优点：

1. 一致性的文件格式：

在SQLite的官方文档中是这样解释的，我们不要将SQLite与Oracle或PostgreSQL去比较，而是应该将它看做fopen和fwrite。与我们自定义格式的数据文件相比，SQLite不仅提供了很好的移植性，如大端小端、32/64位等平台相关问题，而且还提供了数据访问的高效性，如基于某些信息建立索引，从而提高访问或排序该类数据的性能，SQLite提供的事务功能，也是在操作普通文件时无法有效保证的。

2. 在嵌入式或移动设备上的应用：

由于SQLite在运行时占用的资源较少，而且无需任何管理开销，因此对于PDA、智能手机等移动设备来说，SQLite的优势毋庸置疑。

3. 内部数据库：

在有些应用场景中，我们需要为插入到数据库服务器中的数据进行数据过滤或数据清理，以保证最终插入到数据库服务器中的数据有效性。有的时候，数据是否有效，不能通过单一一条记录来进行判断，而是需要和之前一小段时间的历史数据进行特殊的计算，再通过计算的结果判断当前的数据是否合法。在这种应用中，我们可以用SQLite缓冲这部分历史数据。还有一种简单的场景也适用于SQLite，即统计数据的预计算。比如我们正在运行数据实时采集

的服务程序，我们可能需要将每10秒的数据汇总后，形成每小时的统计数据，该统计数据可以极大的减少用户查询时的数据量，从而大幅提高前端程序的查询效率。在这种应用中，我们可以将1小时内的采集数据均缓存在SQLite中，在达到整点时，计算缓存数据后清空该数据。

4. 数据分析：

可以充分利用SQLite提供SQL特征，完成简单的数据统计分析的功能。这一点是CSV文件无法比拟的。

5. 产品Demo和测试：

在需要给客户进行Demo时，可以使用SQLite作为我们的后台数据库，和其他关系型数据库相比，使用SQLite减少了大量的系统部署时间。对于产品的功能性测试而言，SQLite也可以起到相同的作用。

三、和RDBMS相比SQLite的一些劣势：

1. C/S应用：

如果你有多个客户端需要同时访问数据库中的数据，特别是他们之间的数据操作是需要通过网络传输来完成的。在这种情况下，不应该选择SQLite。由于SQLite的数据管理机制更多的依赖于OS的文件系统，因此在这种操作下其效率较低。

2. 数据量较大：

受限于操作系统的文件系统，在处理大数据量时，其效率较低。对于超大数据量的存储，甚至不能提供支持。

3. 高并发：

由于SQLite仅仅提供了粒度很粗的数据锁，如读写锁，因此在每次加锁操作中都会有大量的数据被锁住，即使仅有极小部分的数据会被访问。换句话说，我们可以认为SQLite只是提供了表级锁，没有提供行级锁。在这种同步机制下，并发性能很难高效。

四、个性化特征：

1. 零配置：

SQLite本身并不需要任何初始化配置文件，也没有安装和卸载的过程。当然也不存在服务器实例的启动和停止。在使用的过程中，也无需创建用户和划分权限。在系统出现灾难时，如电源问题、主机问题等，对于SQLite而言，不需要做任何操作。

2. 没有独立的服务器：

和其他关系型数据库不同的是，SQLite没有单独的服务器进程，以供客户端程序访问并提供相关的服务。SQLite作为一种嵌入式数据库，其运行环境与主程序位于同一进程空间，因此它们之间的通信完全是进程内通信，而相比于进程间通信，其效率更高。然而需要特别指出

的是，该种结构在实际运行时确实存在保护性较差的问题，比如此时，应用程序出现问题导致进程崩溃，由于SQLite与其所依赖的进程位于同一进程空间，那么此时SQLite也将随之退出。但是对于独立的服务器进程，则不会有此问题，它们将在密闭性更好的环境下完成它们的工作。

3. 单一磁盘文件：

SQLite的数据库被存放在文件系统的单一磁盘文件内，只要有权限便可随意访问和拷贝，这样带来的主要好处是便于携带和共享。其他的数据库引擎，基本都会将数据库存放在一个磁盘目录下，然后由该目录下的一组文件构成该数据库的数据文件。尽管我们可以直接访问这些文件，但是我们的程序却无法操作它们，只有数据库实例进程才可以做到。这样的好处是带来了更高的安全性和更好的性能，但是也付出了安装和维护复杂的代价。

4. 平台无关性：

这一点在前面已经解释过了。和SQLite相比，很多数据库引擎在备份数据时不能通过该方式直接备份，只能通过数据库系统提供的各种dump和restore工具，将数据库中的数据先导出到本地文件中，之后在load到目标数据库中。这种方式存在显而易见的效率问题，首先需要导出到另外一个文件，如果数据量较大，导出的过程将会比较耗时。然而这只是该操作的一小部分，因为数据导入往往需要更多的时间。数据在导入时需要很多的验证过程，在存储时，也并非简简单单的顺序存储，而是需要按照一定的数据结构、算法和策略存放在不同的文件位置。因此和直接拷贝数据库文件相比，其性能是非常拙劣的。

5. 弱类型：

和大多数支持静态类型的数据库不同的是，SQLite中的数据类型被视为数值的一个属性。因此对于一个数据表列而言，即便在声明该表时给出了该列的类型，我们在插入数据时仍然可以插入任意类型，比如Integer的列被存入字符串'hello'。针对该特征唯一的例外是整型的主键列，对于此种情况，我们只能在该列中存储整型数据。

6. SQL语句编译成虚拟机代码：

很多数据库产品会将SQL语句解析成复杂的，相互嵌套的数据结构，之后再交予执行器遍历该数据结构完成指定的操作。相比于此，SQLite会将SQL语句先编译成字节码，之后再交由其自带的虚拟机去执行。该方式提供了更好的性能和更出色的调试能力。

SQLite学习手册(C/C++接口简介)

一、概述：

在SQLite提供的C/C++接口中，其中5个APIs属于核心接口。在这篇博客中我们将主要介绍它们的用法，以及它们所涉及到的核心SQLite对象，如database_connection和prepared_statement。相比于其它数据库引擎提供的APIs，如OCI、MySQL API等，SQLite提供的接口还是非常易于理解和掌握的。

二、核心对象和接口：

1. 核心对象：

在SQLite中最主要的两个对象是，database_connection和prepared_statement。database_connection对象是由sqlite3_open()接口函数创建并返回的，在应用程序使用任何其他SQLite接口函数之前，必须先调用该函数以便获得database_connnection对象，在随后的其他APIs调用中，都需要该对象作为输入参数以完成相应的工作。至于prepare_statement，我们可以简单的将它视为编译后的SQL语句，因此，所有和SQL语句执行相关的函数也都需要该对象作为输入参数以完成指定的SQL操作。

2. 核心接口：

1). sqlite3_open

上面已经提到过这个函数了，它是操作SQLite数据库的入口函数。该函数返回的database_connection对象是很多其他SQLite APIs的句柄参数。注意，我们通过该函数既可以打开已经存在的数据库文件，也可以创建新的数据库文件。对于该函数返回的database_connection对象，我们可以在多个线程之间共享该对象的指针，以便完成和数据库相关的任意操作。然而在多线程情况下，我们更为推荐的使用方式是，为每个线程创建独立的database_connection对象。对于该函数还有一点也需要额外说明，我们没有必要为了访问多个数据库而创建多个数据库连接对象，因为通过SQLite自带的ATTACH命令可以在一个连接中方便的访问多个数据库。

2). sqlite3_prepare

该函数将SQL文本转换为prepared_statement对象，并在函数执行后返回该对象的指针。事实上，该函数并不会评估参数指定SQL语句，它仅仅是将SQL文本初始化为待执行的状态。最后需要指出的，对于新的应用程序我们可以使用sqlite3_prepare_v2接口函数来替代该函数以完成相同的工作。

3). sqlite3_step

该函数用于评估sqlite3_prepare函数返回的prepared_statement对象，在执行完该函数之后，prepared_statement对象的内部指针将指向其返回的结果集的第一行。如果打算进一步迭代其后的数据行，就需要不断的调用该函数，直到所有的数据行都遍历完毕。然而对于INSERT、UPDATE和DELETE等DML语句，该函数执行一次即可完成。

4). sqlite3_column

该函数用于获取当前行指定列的数据，然而严格意义上讲，此函数在SQLite的接口函数中并不存在，而是由一组相关的接口函数来完成该功能，其中每个函数都返回不同类型的数据，如：

- sqlite3_column_blob
- sqlite3_column_bytes
- sqlite3_column_bytes16
- sqlite3_column_double
- sqlite3_column_int
- sqlite3_column_int64
- sqlite3_column_text
- sqlite3_column_text16
- sqlite3_column_type
- sqlite3_column_value
- sqlite3_column_count

其中sqlite3_column_count函数用于获取当前结果集中的字段数据。下面是使用sqlite3_step和sqlite3_column函数迭代结果集中每行数据的伪代码，注意这里作为示例代码简化了对字段类型的判断：

```
1     int fieldCount = sqlite3_column_count(...);
2     while (sqlite3_step(...) <> EOF) {
3         for (int i = 0; i < fieldCount; ++i) {
4             int v = sqlite3_column_int(...,i);
5         }
6     }
```

5). sqlite3_finalize

该函数用于销毁prepared statement对象，否则将会造成内存泄露。

6). sqlite3_close

该函数用于关闭之前打开的database_connection对象，其中所有和该对象相关的prepared_statements对象都必须在此之前先被销毁。

三、参数绑定：

和大多数关系型数据库一样，SQLite的SQL文本也支持变量绑定，以便减少SQL语句被动态解析的次数，从而提高数据查询和数据操作的效率。要完成该操作，我们需要使用SQLite提供的另外两个接口APIs，sqlite3_reset和sqlite3_bind。见如下示例：

```
1     void test_parameter_binding() {
2         //1\. 不带参数绑定的情况下插入多条数据。
3         char strSQL[128];
4         for (int i = 0; i < MAX_ROWS; ++i) {
5             sprintf(strSQL,"insert into testtable values(%d)",i);
6             sqlite3_prepare_v2(..., strSQL);
7             sqlite3_step(prepared_stmt);
8             sqlite3_finalize(prepared_stmt);
9         }
10        //2\. 参数绑定的情况下插入多条数据。
11        string strSQLWithParameter = "insert into testtable values(?)";
12        sqlite3_prepare_v2(..., strSQL);
13        for (int i = 0; i < MAX_ROWS; ++i) {
14            sqlite3_bind(...,i);
15            sqlite3_step(prepared_stmt);
16            sqlite3_reset(prepared_stmt);
17        }
18        sqlite3_finalize(prepared_stmt);
19    }
```

这里首先需要说明的是，SQL语句"insert into testtable values(?)"中的问号(?)表示参数变量的占位符，该规则在很多关系型数据库中都是一致的，因此这对于数据库移植操作还是比较方便的。

通过上面的示例代码可以显而易见的看出，参数绑定写法的执行效率要高于每次生成不同的SQL语句的写法，即2)在效率上要明显优于1)，下面是针对这两种写法的具体比较：

1). 单单从程序表面来看，前者在for循环中执行了更多的任务，比如字符串的填充、SQL语句的prepare，以及prepared_statement对象的释放。

2). 在SQLite的官方文档中明确的指出，sqlite3_prepare_v2的执行效率往往要低于sqlite3_step的效率。

3). 当插入的数据量较大时，后者带来的效率提升还是相当可观的。

SQLite学习手册(数据表和视图)

一、创建数据表:

该命令的语法规则和使用方式与大多数关系型数据库基本相同，因此我们还是以示例的方式来演示SQLite中创建表的各种规则。但是对于一些SQLite特有的规则，我们会给予额外的说明。注：以下所有示例均是在sqlite自带命令行工具中完成的。

1). 最简单的数据表：

```
sqlite> CREATE TABLE testtable (first_col integer);
```

这里需要说明的是，对于自定义数据表表名，如testtable，不能以sqlite_开头，因为以该前缀定义的表名都用于sqlite内部。

2). 创建带有缺省值的数据表：

```
sqlite> CREATE TABLE testtable (first_col integer DEFAULT 0, second_col varchar DEFAULT '
```

3). 在指定数据库创建表：

```
sqlite> ATTACH DATABASE 'd:/mydb.db' AS mydb;
sqlite> CREATE TABLE mydb.testtable (first_col integer);
```

这里先通过ATTACH DATABASE命令将一个已经存在的数据库文件attach到当前的连接中，之后再通过指定数据库名的方式在目标数据库中创建数据表，如mydb.testtable。关于该规则还需要给出一些额外的说明，如果我们在创建数据表时没有指定数据库名，那么将会在当前连接的main数据库中创建该表，在一个连接中只能有一个main数据库。如果需要创建临时表，就无需指定数据库名，见如下示例：

--创建两个表，一个临时表和普通表。

```
sqlite> CREATE TEMP TABLE temptable(first_col integer);
sqlite> CREATE TABLE testtable (first_col integer);
```

--将当前连接中的缓存数据导出到本地文件，同时退出当前连接。

```
sqlite> backup d:/mydb.db
sqlite> exit
```

--重新建立sqlite的连接，并将刚刚导出的数据库作为主库重新导入。

```
sqlite> restore d:/mydb.db
```

--查看该数据库中的表信息，通过结果可以看出临时表并没有被持久化到数据库文件中。

```
sqlite> .tables
testtable
```

4). "IF NOT EXISTS"从句：

如果当前创建的数据表名已经存在，即与已经存在的表名、视图名和索引名冲突，那么本次创建操作将失败并报错。然而如果在创建表时加上"IF NOT EXISTS"从句，那么本次创建操作将不会有任何影响，即不会有错误抛出，除非当前的表名和某一索引名冲突。

```
sqlite> CREATE TABLE testtable (first_col integer);
Error: table testtable already exists
sqlite> CREATE TABLE IF NOT EXISTS testtable (first_col integer);
```

5). CREATE TABLE ... AS SELECT：

通过该方式创建的数据表将与SELECT查询返回的结果集具有相同的Schema信息，但是不包含缺省值和主键等约束信息。然而新创建的表将会包含结果集返回的所有数据。

```
sqlite> CREATE TABLE testtable2 AS SELECT * FROM testtable;
sqlite> .schema testtable2
CREATE TABLE testtable2(first_col INT);
```

.schema命令是sqlite3命令行工具的内置命令，用于显示当前数据表的CREATE TABLE语句。

6). 主键约束：

--直接在字段的定义上指定主键。

```
sqlite> CREATE TABLE testtable (first_col integer PRIMARY KEY ASC
```

--在所有字段已经定义完毕后，再定义表的数约束，这里定义的是基于first_col和second_col的联合主键。

```
sqlite> CREATE TABLE testtable2 (
...>     first_col integer,
...>     second_col integer,
...>     PRIMARY KEY (first_col,second_col)
...> );
```

和其他关系型数据库一样，主键必须是唯一的。

7). 唯一性约束：

--直接在字段的定义上指定唯一性约束。

```
sqlite> CREATE TABLE testtable (first_col integer UNIQUE);
```

--在所有字段已经定义完毕后，在定义表的唯一性约束，这里定义的是基于两个列的唯一性约束。

```
sqlite> CREATE TABLE testtable2 (
...>     first_col integer,
...>     second_col integer,
...>     UNIQUE (first_col,second_col)
...> );
```

在SQLite中，NULL值被视为和其他任何值都是不同的，这样包括和其他的NULL值，如下例：

```
sqlite> DELETE FROM testtable;
sqlite> SELECT count(*) FROM testtable;
count(*)
----------
0
sqlite> INSERT INTO testtable VALUES(NULL);
sqlite> INSERT INTO testtable VALUES(NULL);
sqlite> SELECT count(*) FROM testtable;
count(*)
----------
2
```

由此可见，两次插入的NULL值均插入成功。

8). 为空(NOT NULL)约束：

```
sqlite> CREATE TABLE testtable(first_col integer NOT NULL);
sqlite> INSERT INTO testtable VALUES(NULL);
Error: testtable.first_col may not be NULL
```

从输出结果可以看出，first_col已经被定义了非空约束，因此不能在插入NULL值了。

9). 检查性约束：

```
sqlite> CREATE TABLE testtable (first_col integer CHECK (first_col < 5));
sqlite> INSERT INTO testtable VALUES(4);
sqlite> INSERT INTO testtable VALUES(20); _-- 20违反了字段first_col的检查性约束(first_col < 5
Error: constraint failed
```

--和之前的其它约束一样，检查性约束也是可以基于表中的多个列来定义的。

```
sqlite> CREATE TABLE testtable2 (
...>     first_col integer,
...>     second_col integer,
...>     CHECK (first_col > 0 AND second_col < 0)
...> );
```

二、表的修改：

SQLite对ALTER TABLE命令支持的非常有限，仅仅是修改表名和添加新字段。其它的功能，如重命名字段、删除字段和添加删除约束等均为提供支持。

1). 修改表名：

需要先说明的是，SQLite中表名的修改只能在同一个数据库中，不能将其移动到Attached数据库中。再有就是一旦表名被修改后，该表已存在的索引将不会受到影响，然而依赖该表的视图和触发器将不得不重新修改其定义。

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> ALTER TABLE testtable RENAME TO testtable2;
sqlite> .tables
testtable2
```

通过.tables命令的输出可以看出，表testtable已经被修改为testtable2。

2). 新增字段：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> ALTER TABLE testtable ADD COLUMN second_col integer;
sqlite> .schema testtable
CREATE TABLE "testtable" (first_col integer, second_col integer);
```

通过.schema命令的输出可以看出，表testtable的定义中已经包含了新增字段。

关于ALTER TABLE最后需要说明的是，在SQLite中该命令的执行时间是不会受到当前表行数的影响，也就是说，修改有一千万行数据的表和修改只有一条数据的表所需的时间几乎是相等的。

三、表的删除：

在SQLite中如果某个表被删除了，那么与之相关的索引和触发器也会被随之删除。在很多其他的关系型数据库中是不可以这样的，如果必须要删除相关对象，只能在删除表语句中加入WITH CASCADE从句。见如下示例：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> DROP TABLE testtable;
sqlite> DROP TABLE testtable;
Error: no such table: testtable
sqlite> DROP TABLE IF EXISTS testtable;
```

从上面的示例中可以看出，如果删除的表不存在，SQLite将会报错并输出错误信息。如果希望在执行时不抛出异常，我们可以添加IF EXISTS从句，该从句的语义和CREATE TABLE中的完全相同。

四、创建视图：

我们这里只是给出简单的SQL命令示例，具体的含义和技术细节可以参照上面的创建数据表部分，如临时视图、"IF NOT EXISTS"从句等。

1). 最简单的视图：

```
sqlite> CREATE VIEW testview AS SELECT * FROM testtable WHERE first_col > 100;
```

2). 创建临时视图：

```
sqlite> CREATE TEMP VIEW tempview AS SELECT * FROM testtable WHERE first_col > 100;
```

3). "IF NOT EXISTS"从句：

```
sqlite> CREATE VIEW testview AS SELECT * FROM testtable WHERE first_col > 100;
Error: table testview already exists
sqlite> CREATE VIEW IF NOT EXISTS testview AS SELECT * FROM testtable WHERE first_col > 1
```

五、删除视图：

该操作的语法和删除表基本相同，因此这里只是给出示例：

```
sqlite> DROP VIEW testview;
sqlite> DROP VIEW testview;
Error: no such view: testview
sqlite> DROP VIEW IF EXISTS testview;
```

SQLite学习手册(内置函数)

一、聚合函数：

SQLite中支持的聚合函数在很多其他的关系型数据库中也同样支持，因此我们这里将只是给出每个聚集函数的简要说明，而不在给出更多的示例了。这里还需要进一步说明的是，对于所有聚合函数而言，distinct关键字可以作为函数参数字段的前置属性，以便在进行计算时忽略到所有重复的字段值，如count(distinct x)。

函数	说明
avg(x)	该函数返回在同一组内参数字段的平均值。对于不能转换为数字值的String和BLOB类型的字段值，如'HELLO'，SQLite会将其视为0。avg函数的结果总是浮点型，唯一的例外是所有的字段值均为NULL，那样该函数的结果也为NULL。
count(x|*)	count(x)函数返回在同一组内，x字段中值不等于NULL的行数。count(*)函数返回在同一组内的数据行数。
group_concat(x[,y])	该函数返回一个字符串，该字符串将会连接所有非NULL的x值。该函数的y参数将作为每个x值之间的分隔符，如果在调用时忽略该参数，在连接时将使用缺省分隔符","。再有就是各个字符串之间的连接顺序是不确定的。
max(x)	该函数返回同一组内的x字段的最大值，如果该字段的所有值均为NULL，该函数也返回NULL。
min(x)	该函数返回同一组内的x字段的最小值，如果该字段的所有值均为NULL，该函数也返回NULL。
sum(x)	该 函数返回同一组内的x字段值的总和，如果字段值均为NULL，该函数也返回NULL。如果所有的x字段值均为整型或者NULL，该函数返回整型值，否则就 返回浮点型数值。最后需要指出的是，如果所有的数据值均为整型，一旦结果超过上限时将会抛出"integer overflow"的异常。
total(x)	该函数不属于标准SQL，其功能和sum基本相同，只是计算结果比sum更为合理。比如当所有字段值均为NULL时，和sum不同的是，该函数返回0.0。再有就是该函数始终返回浮点型数值。该函数始终都不会抛出异常。

二、核心函数：

以下函数均为SQLite缺省提供的内置函数，其声明和描述见如下列表：

函数	说明
abs(X)	该函数返回数值参数X的绝对值，如果X为NULL，则返回NULL，如果X为不能转换成数值的字符串，则返回0，如果X值超出Integer的上限，则抛出"Integer Overflow"的异常。

changes()	该函数返回最近执行的INSERT、UPDATE和DELETE语句所影响的数据行数。我们也可以通过执行C/C++函数sqlite3_changes()得到相同的结果。
coalesce(X,Y,...)	返回函数参数中第一个非NULL的参数，如果参数都是NULL，则返回NULL。该函数至少2个参数。
ifnull(X,Y)	该函数等同于两个参数的coalesce()函数，即返回第一个不为NULL的函数参数，如果两个均为NULL，则返回NULL。
length(X)	如果参数X为字符串，则返回字符的数量，如果为数值，则返回该参数的字符串表示形式的长度，如果为NULL，则返回NULL。
lower(X)	返回函数参数X的小写形式，缺省情况下，该函数只能应用于ASCII字符。
ltrim(X[,Y])	如果没有可选参数Y，该函数将移除参数X左侧的所有空格符。如果有参数Y，则移除X左侧的任意在Y中出现的字符。最后返回移除后的字符串。
max(X,Y,...)	返回函数参数中的最大值，如果有任何一个参数为NULL，则返回NULL。
min(X,Y,...)	返回函数参数中的最小值，如果有任何一个参数为NULL，则返回NULL。
nullif(X,Y)	如果函数参数相同，返回NULL，否则返回第一个参数。
random()	返回整型的伪随机数。
replace(X,Y,Z)	将字符串类型的函数参数X中所有子字符串Y替换为字符串Z，最后返回替换后的字符串，源字符串X保持不变。
round(X[,Y])	返回数值参数X被四舍五入到Y刻度的值，如果参数Y不存在，缺省参数值为0。
rtrim(X[,Y])	如果没有可选参数Y，该函数将移除参数X右侧的所有空格符。如果有参数Y，则移除X右侧的任意在Y中出现的字符。最后返回移除后的字符串。
substr(X,Y[,Z])	返回函数参数X的子字符串，从第Y位开始(X中的第一个字符位置为1)截取Z长度的字符，如果忽略Z参数，则取第Y个字符后面的所有字符。如果Z的值为负数，则从第Y位开始，向左截取abs(Z)个字符。如果Y值为负数，则从X字符串的尾部开始计数到第abs(Y)的位置开始。
total_changes()	该函数返回自从该连接被打开时起，INSERT、UPDATE和DELETE语句总共影响的行数。我们也可以通过C/C++接口函数sqlite3_total_changes()得到相同的结果。
trim(x[,y])	如果没有可选参数Y，该函数将移除参数X两侧的所有空格符。如果有参数Y，则移除X两侧的任意在Y中出现的字符。最后返回移除后的字符串。
upper(X)	返回函数参数X的大写形式，缺省情况下，该函数只能应用于ASCII字符。
	返回函数参数数据类型的字符串表示形式，如"Integer、text、

	real、null"等。

三、日期和时间函数：

SQLite主要支持以下四种与日期和时间相关的函数，如：

1). `date(timestring, modifier, modifier, ...)`

2). `time(timestring, modifier, modifier, ...)`

3). `datetime(timestring, modifier, modifier, ...)`

4). `strftime(format, timestring, modifier, modifier, ...)`

以上所有四个函数都接受一个时间字符串作为参数，其后再跟有0个或多个修改符。其中strftime()函数还接受一个格式字符串作为其第一个参数。strftime()和C运行时库中的同名函数完全相同。至于其他三个函数，date函数的缺省格式为："YYYY-MM-DD"，time函数的缺省格式为："HH:MM:SS"，datetime函数的缺省格式为："YYYY-MM-DD HH:MM:SS"。

1. strftime函数的格式信息：

格式	说明
%d	day of month: 00
%f	fractional seconds: SS.SSS
%H	hour: 00-24
%j	day of year: 001-366
%J	Julian day number
%m	month: 01-12
%M	minute: 00-59
%s	seconds since 1970-01-01
%S	seconds: 00-59
%w	day of week 0-6 with Sunday==0
%W	week of year: 00-53
%Y	year: 0000-9999
%%	%

需要额外指出的是，其余三个时间函数均可用strftime来表示，如：

```
date(...)         strftime('%Y-%m-%d', ...)
time(...)         strftime('%H:%M:%S', ...)
datetime(...)   strftime('%Y-%m-%d %H:%M:%S', ...)
```

2. 时间字符串的格式：

见如下列表：

1). YYYY-MM-DD

2). YYYY-MM-DD HH:MM

3). YYYY-MM-DD HH:MM:SS

4). YYYY-MM-DD HH:MM:SS.SSS

5). HH:MM

6). HH:MM:SS

7). HH:MM:SS.SSS

8). now

5)到7)中只是包含了时间部分，SQLite将假设日期为2000-01-01。8)表示当前时间。

3. 修改符：

见如下列表：

1). NNN days

2). NNN hours

3). NNN minutes

4). NNN.NNNN seconds

5). NNN months

6). NNN years

7). start of month

8). start of year

9). start of day

10).weekday N

1)到6)将只是简单的加减指定数量的日期或时间值，如果NNN的值为负数，则减，否则加。7)到9)则将时间串中的指定日期部分设置到当前月、年或日的开始。10)则将日期前进到下一个星期N，其中星期日为0。注：修改符的顺序极为重要，SQLite将会按照从左到右的顺序依次执行修改符。

4. 示例：

--返回当前日期。

```
sqlite> SELECT date('now');
2012-01-15
```

--返回当前月的最后一天。

```
sqlite> SELECT date('now','start of month','1 month','-1 day');
2012-01-31
```

--返回从1970-01-01 00:00:00到当前时间所流经的秒数。

```
sqlite> SELECT strftime('%s','now');
1326641166
```

--返回当前年中10月份的第一个星期二是日期。

```
sqlite> SELECT date('now','start of year','+9 months','weekday 2');
2012-10-02
```

SQLite学习手册(索引和数据分析/清理)

一、创建索引：

在SQLite中，创建索引的SQL语法和其他大多数关系型数据库基本相同，因为这里也仅仅是给出示例用法：

```
sqlite> CREATE TABLE testtable (first_col integer,second_col integer);
```

--创建最简单的索引，该索引基于某个表的一个字段。

```
sqlite> CREATE INDEX testtable_idx ON testtable(first_col);
```

--创建联合索引，该索引基于某个表的多个字段，同时可以指定每个字段的排序规则(升序/降序)。

```
sqlite> CREATE INDEX testtable_idx2 ON testtable(first_col ASC,second_col DESC);
```

--创建唯一性索引，该索引规则和数据表的唯一性约束的规则相同，即NULL和任何值都不同，包括NULL本身。

```
sqlite> CREATE UNIQUE INDEX testtable_idx3 ON testtable(second_col DESC);
sqlite> .indices testtable
testtable_idx
testtable_idx2
testtable_idx3
```

从.indices命令的输出可以看出，三个索引均已成功创建。

二、删除索引：

索引的删除和视图的删除非常相似，含义也是如此，因此这里也只是给出示例：

```
sqlite> DROP INDEX testtable_idx;
```

--如果删除不存在的索引将会导致操作失败，如果在不确定的情况下又不希望错误被抛出，可以使用"IF EXISTS"从句_。

```
sqlite> DROP INDEX testtable_idx;
Error: no such index: testtable_idx
sqlite> DROP INDEX IF EXISTS testtable_idx;
```

三、重建索引：

重建索引用于删除已经存在的索引，同时基于其原有的规则重建该索引。这里需要说明的是，如果在REINDEX语句后面没有给出数据库名，那么当前连接下所有Attached数据库中所有索引都会被重建。如果指定了数据库名和表名，那么该表中的所有索引都会被重建，如果只是指定索引名，那么当前数据库的指定索引被重建。

--当前连接attached所有数据库中的索引都被重建。

```
sqlite> REINDEX;
```

--重建当前主数据库中testtable表的所有索引。

```
sqlite> REINDEX testtable;
```

--重建当前主数据库中名称为testtable_idx2的索引。

```
sqlite> REINDEX testtable_idx2;
```

四、数据分析：

和PostgreSQL非常相似，SQLite中的ANALYZE命令也同样用于分析数据表和索引中的数据，并将统计结果存放于SQLite的内部系统表中，以便于查询优化器可以根据分析后的统计数据选择最优的查询执行路径，从而提高整个查询的效率。见如下示例：

--如果在ANALYZE命令之后没有指定任何参数，则分析当前连接中所有Attached数据库中的表和索引。

```
sqlite> ANALYZE;
```

--如果指定数据库作为ANALYZE的参数，那么该数据库下的所有表和索引都将被分析并生成统计数据。

```
sqlite> ANALYZE main;
```

--如果指定了数据库中的某个表或索引为ANALYZE的参数，那么该表和其所有关联的索引都将被分析。

```
sqlite> ANALYZE main.testtable;
sqlite> ANALYZE main.testtable_idx2;
```

五、数据清理：

和PostgreSQL中的VACUUM命令相比，他们的功能以及实现方式非常相似，不同的是PostgreSQL提供了更细的粒度，而SQLite只能将该命令作用于数据库，无法再精确到数据库中指定的数据表或者索引，然而这一点恰恰是PostgreSQL可以做到的。

当某个数据库中的一个或多个数据表存在大量的插入、更新和删除等操作时，将会有大量的磁盘空间被已删除的数据所占用，在没有执行VACUUM命令之前，SQLite并没有将它们归还于操作系统。由于该类数据表中的数据存储非常分散，因此在查询时，无法得到更好的批量IO读取效果，从而影响了查询效率。

在SQLite中，仅支持清理当前连接中的主数据库，而不能清理其它Attached数据库。VACUUM命令在完成数据清理时采用了和PostgreSQL相同的策略，即创建一个和当前数据库文件相同大小的新数据库文件，之后再将该数据库文件中的数据有组织的导入到新文件中，其中已经删除的数据块将不会被导入，在完成导入后，收缩新数据库文件的尺寸到适当的大小。该命令的执行非常简单，如：

```
sqlite> VACUUM;
```

SQLite学习手册(数据库和事务)

一、Attach数据库：

ATTACH DATABASE语句添加另外一个数据库文件到当前的连接中，如果文件名为":memory:"，我们可以将其视为内存数据库，内存数据库无法持久化到磁盘文件上。如果操作Attached数据库中的表，则需要在表名前加数据库名，如dbname.table_name。最后需要说明的是，如果一个事务包含多个Attached数据库操作，那么该事务仍然是原子的。见如下示例：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> INSERT INTO testtable VALUES(1);
sqlite> .backup 'D:/mydb.db'   _--将当前连接中的主数据库备份到指定文件。_
sqlite> .exit
```

--重新登录sqlite命令行工具：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> INSERT INTO testtable VALUES(2);
sqlite> INSERT INTO testtable VALUES(1);
sqlite> ATTACH DATABASE 'D:/mydb.db' AS mydb;
sqlite> .header on            --查询结果将字段名作为标题输出。
sqlite> .mode column        --将每列都分开显示。
sqlite> SELECT t1.first_col FROM testtable t1, mydb.testtable t2 WHERE t.first_col = t2.f
first_col
----------
1
```

二、Detach数据库：

卸载将当前连接中的指定数据库，注意main和temp数据库无法被卸载。见如下示例：

--该示例承载上面示例的结果，即mydb数据库已经被Attach到当前的连接中。

```
sqlite> DETACH DATABASE mydb;
sqlite> SELECT t1.first_col FROM testtable t1, mydb.testtable t2 WHERE t.first_col = t2.f
Error: no such table: mydb.testtable
```

三、事务：

在SQLite中，如果没有为当前的SQL命令(SELECT除外)显示的指定事务，那么SQLite会自动为该操作添加一个隐式的事务，以保证该操作的原子性和一致性。当然，SQLite也支持显示的事务，其语法与大多数关系型数据库相比基本相同。见如下示例：

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO testtable VALUES(1);
sqlite> INSERT INTO testtable VALUES(2);
sqlite> COMMIT TRANSACTION;      --显示事务被提交，数据表中的数据也发生了变化。
sqlite> SELECT COUNT(*) FROM testtable;
COUNT(*)
----------
2
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO testtable VALUES(1);
sqlite> ROLLBACK TRANSACTION;  --显示事务被回滚，数据表中的数据没有发生变化。
sqlite> SELECT COUNT(*) FROM testtable;
COUNT(*)
----------
2
```

SQLite学习手册(表达式)

一、常用表达式：

和大多数关系型数据库一样，SQLite能够很好的支持SQL标准中提供的表达式，其函数也与SQL标准保持一致，如：

```
||
*    /    %
+    -
<<   >>   &    |
<    <=   >    >=
=    ==   !=   <>   IS   IS NOT   IN   LIKE
AND
OR
~    NOT
```

在上面的表达式中，唯一需要说明的是"||"，该表达式主要用于两个字符串之间的连接，其返回值为连接后的字符串，即便该操作符两边的操作数为非字符串类型，在执行该表达式之前都需要被提前转换为字符串类型，之后再进行连接。

二、条件表达式：

该表达式的语法规则如下：

```
1). CASE x WHEN w1 THEN r1 WHEN w2 THEN r2 ELSE r3 END
2). CASE WHEN x=w1 THEN r1 WHEN x=w2 THEN r2 ELSE r3 END
```

对于第一种情况，条件表达式x只需计算一次，然后分别和WHEN关键字后的条件逐一进行比较，直到找到相等的条件，其比较规则等价于等号(=)表达式。如果找到匹配的条件，则返回其后THEN关键字所指向的值，如果没有找到任何匹配，则返回ELSE关键字之后的值，如果不存在ELSE分支，则返回NULL。对于第二种情况，和第一种情况相比，唯一的差别就是表达式x可能被多次执行，比如第一个WHEN条件不匹配，则继续计算后面的WHEN条件，其它规则均与第一种完全相同。最后需要说明的是，以上两种形式的CASE表达式均遵守短路原则，即第一个表达式的条件一旦匹配，其后所有的WHEN表达式均不会再被执行或比较。

三、转换表达式：

该表达式的语法规则如下：

```
CAST(expr AS target_type)
```

该表达式会将参数expr转换为target_type类型，具体的转换规则见如下列表：

目标类型	转换规则描述
TEXT	如果转换INTEGER或REAL类型的值到TEXT类型直接转换即可，就像C/C++接口函数sqlite3_snprintf所完成的工作。
REAL	如果转换TEXT类型的值到REAL类型，在该文本的最前部，将可以转换为实数的文本转换为相应的实数，其余部分忽略。其中该文本值的前导零亦将被全部忽略。如果该文本值没有任何字符可以转换为实数，CAST表达式的转换结果为0.0。
INTEGER	如果转换TEXT类型的值到INTEGER类型，在该文本的最前部，将可以转换为整数的文本转换为相应的整数，其余部分忽略。其中该文本值的前导零亦将被全部忽略。如果该文本值没有任何字符可以转换为整数，CAST表达式的转换结果为0。如果转换将一个实数值转换为INTEGER类型，则直接截断实数小数部分。如果实数过大，则返回最大的负整数：-9223372036854775808。
NUMERIC	如果转换文本值到NUMERIC类型，则先将该值强制转换为REAL类型，只有在将REAL转换为INTEGER不会导致数据信息丢失以及完全可逆的情况下，SQLite才会进一步将其转换为INTEGER类型。

最后需要说明的是，如果expr为NULL，则转换的结果也为NULL。

SQLite学习手册(数据类型)

一、存储种类和数据类型：

SQLite将数据值的存储划分为以下几种存储类型：

NULL: 表示该值为NULL值。

INTEGER: 无符号整型值。

REAL: 浮点值。

TEXT: 文本字符串，存储使用的编码方式为UTF-8、UTF-16BE、UTF-16LE。

BLOB: 存储Blob数据，该类型数据和输入数据完全相同。

由于SQLite采用的是动态数据类型，而其他传统的关系型数据库使用的是静态数据类型，即字段可以存储的数据类型是在表声明时即以确定的，因此它们之间在数据存储方面还是存在着很大的差异。在SQLite中，存储分类和数据类型也有一定的差别，如INTEGER存储类别可以包含6种不同长度的Integer数据类型，然而这些INTEGER数据一旦被读入到内存后，SQLite会将其全部视为占用8个字节无符号整型。因此对于SQLite而言，即使在表声明中明确了字段类型，我们仍然可以在该字段中存储其它类型的数据。然而需要特别说明的是，尽管SQLite为我们提供了这种方便，但是一旦考虑到数据库平台的可移植性问题，我们在实际的开发中还是应该尽可能的保证数据类型的存储和声明的一致性。除非你有极为充分的理由，同时又不再考虑数据库平台的移植问题，在此种情况下确实可以使用SQLite提供的此种特征。

1. 布尔数据类型：

SQLite并没有提供专门的布尔存储类型，取而代之的是存储整型1表示true，0表示false。

2. 日期和时间数据类型：

和布尔类型一样，SQLite也同样没有提供专门的日期时间存储类型，而是以TEXT、REAL和INTEGER类型分别不同的格式表示该类型，如：

TEXT: "YYYY-MM-DD HH:MM:SS.SSS"

REAL: 以Julian日期格式存储

INTEGER: 以Unix时间形式保存数据值，即从1970-01-01 00:00:00到当前时间所流经的秒数。

二、类型亲缘性：

为了最大化SQLite和其它数据库引擎之间的数据类型兼容性，SQLite提出了"类型亲缘性(Type Affinity)"的概念。我们可以这样理解"类型亲缘性 "，在表字段被声明之后，SQLite都会根据该字段声明时的类型为其选择一种亲缘类型，当数据插入时，该字段的数据将会优先采用亲缘类型作为该值的存储方式，除非亲缘类型不匹配或无法转换当前数据到该亲缘类型，这样SQLite才会考虑其它更适合该值的类型存储该值。SQLite目前的版本支持以下五种亲缘类型：

亲缘类型	描述
TEXT	数值型数据在被插入之前，需要先被转换为文本格式，之后再插入到目标字段中。
NUMERIC	当文本数据被插入到亲缘性为NUMERIC的字段中时，如果转换操作不会导致数据信息丢失以及完全可逆，那么SQLite就会将该文本数据转换为INTEGER或REAL类型的数据，如果转换失败，SQLite仍会以TEXT方式存储该数据。对于NULL或BLOB类型的新数据，SQLite将不做任何转换，直接以NULL或BLOB的方式存储该数据。需要额外说明的是，对于浮点格式的常量文本，如"30000.0"，如果该值可以转换为INTEGER同时又不会丢失数值信息，那么SQLite就会将其转换为INTEGER的存储方式。
INTEGER	对于亲缘类型为INTEGER的字段，其规则等同于NUMERIC，唯一差别是在执行CAST表达式时。
REAL	其规则基本等同于NUMERIC，唯一的差别是不会将"30000.0"这样的文本数据转换为INTEGER存储方式。
NONE	不做任何的转换，直接以该数据所属的数据类型进行存储。

1. 决定字段亲缘性的规则：

字段的亲缘性是根据该字段在声明时被定义的类型来决定的，具体的规则可以参照以下列表。需要注意的是以下列表的顺序，即如果某一字段类型同时符合两种亲缘性，那么排在前面的规则将先产生作用。

1). 如果类型字符串中包含"INT"，那么该字段的亲缘类型是INTEGER。

2). 如果类型字符串中包含"CHAR"、"CLOB"或"TEXT"，那么该字段的亲缘类型是TEXT，如VARCHAR。

3). 如果类型字符串中包含"BLOB"，那么该字段的亲缘类型是NONE。

4). 如果类型字符串中包含"REAL"、"FLOA"或"DOUB"，那么该字段的亲缘类型是REAL。

5). 其余情况下，字段的亲缘类型为NUMERIC。

2. 具体示例：

声明类型	亲缘类型	应用规则
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER	1
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT	2
BLOB	NONE	3
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL	4
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC	5

注：在SQLite中，类型VARCHAR(255)的长度信息255没有任何实际意义，仅仅是为了保证与其它数据库的声明一致性。

三、比较表达式：

在SQLite3中支持的比较表达式有："=", "==", "<", "<=", ">", ">=", "!=", "<>", "IN", "NOT IN", "BETWEEN", "IS" and "IS NOT"。

数据的比较结果主要依赖于操作数的存储方式，其规则为：

1). 存储方式为NULL的数值小于其它存储类型的值。

2). 存储方式为INTEGER和REAL的数值小于TEXT或BLOB类型的值，如果同为INTEGER或REAL，则基于数值规则进行比较。

3). 存储方式为TEXT的数值小于BLOB类型的值，如果同为TEXT，则基于文本规则(ASCII值)进行比较。

4). 如果是两个BLOB类型的数值进行比较，其结果为C运行时函数memcmp()的结果。

四、操作符：

所有的数学操作符(+, -, *, /, %, <<, >>, &, and |)在执行之前都会先将操作数转换为NUMERIC存储类型，即使在转换过程中可能会造成数据信息的丢失。此外，如果其中一个操作数为NULL，那么它们的结果亦为NULL。在数学操作符中，如果其中一个操作数看上去并不像数值类型，那么它们结果为0或0.0。

SQLite学习手册(命令行工具)

工欲善其事，必先利其器。学好SQLite的命令行工具，对于我们学习SQLite本身而言是非常非常有帮助的。最基本的一条就是，它让我们学习SQLite的过程更加轻松愉快。言归正传吧，在SQLite的官方下载网站，提供了支持多个平台的命令行工具，使用该工具我们可以完成大多数常用的SQLite操作，就像sqlplus之于Oracle。以下列表给出了该工具的内置命令：

命令名	命令说明
.help	列出所有内置命令。
.backup DBNAME FILE	备份指定的数据库到指定的文件，缺省为当前连接的main数据库。
.databases	列出当前连接中所有attached数据库名和文件名。
.dump TABLENAME ...	以SQL文本的格式DUMP当前连接的main数据库，如果指定了表名，则只是DUMP和表名匹配的数据表。参数TABLENAME支持LIKE表达式支持的通配符。
.echo ON|OFF	打开或关闭显示输出。
.exit	退出当前程序。
.explain ON|OFF	打开或关闭当前连接的SELECT输出到Human Readable形式。
.header(s) ON|OFF	在显示SELECT结果时，是否显示列的标题。
.import FILE TABLE	导入指定文件的数据到指定表。
.indices TABLENAME	显示所有索引的名字，如果指定表名，则仅仅显示匹配该表名的数据表的索引，参数TABLENAME支持LIKE表达式支持的通配符。
.log FILE|off	打开或关闭日志功能，FILE可以为标准输出stdout，或标准错误输出stderr。
.mode MODE TABLENAME	设置输出模式，这里最为常用的模式是column模式，使SELECT输出列左对齐显示。
.nullvalue STRING	使用指定的字符串代替NULL值的显示。
.output FILENAME	将当前命令的所有输出重定向到指定的文件。
.output stdout	将当前命令的所有输出重定向到标准输出(屏幕)。

.quit	退出当前程序。
.read FILENAME	执行指定文件内的SQL语句。
.restore DBNAME FILE	从指定的文件还原数据库，缺省为main数据库，此时也可以指定其它数据库名，被指定的数据库成为当前连接的attached数据库。
.schema TABLENAME	显示数据表的创建语句，如果指定表名，则仅仅显示匹配该表名的数据表创建语句，参数TABLENAME支持LIKE表达式支持的通配符。
.separator STRING	改变输出模式和.import的字段间分隔符。
.show	显示各种设置的当前值。
.tables TABLENAME	列出当前连接中main数据库的所有表名，如果指定表名，则仅仅显示匹配该表名的数据表名称，参数TABLENAME支持LIKE表达式支持的通配符。
.width NUM1 NUM2 ...	在MODE为column时，设置各个字段的宽度，注意：该命令的参数顺序表示字段输出的顺序。

见如下常用示例：

1). 备份和还原数据库。

--在当前连接的main数据库中创建一个数据表，之后再通过.backup命令将main数据库备份到D:/mydb.db文件中。

```
sqlite> CREATE TABLE mytable (first_col integer);
sqlite> .backup 'D:/mydb.db'
sqlite> .exit
```

--通过在命令行窗口下执行sqlite3.exe以重新建立和SQLite的连接。

--从备份文件D:/mydb.db中恢复数据到当前连接的main数据库中，再通过.tables命令可以看到mytable表。

```
sqlite> .restore 'D:/mydb.db'
sqlite> .tables
mytable
```

2). DUMP数据表的创建语句到指定文件。

--先将命令行当前的输出重定向到D:/myoutput.txt，之后在将之前创建的mytable表的声明语句输出到该文件。

```
sqlite> .output D:/myoutput.txt
sqlite> .dump mytabl%
sqlite> .exit
```

--在DOS环境下用记事本打开目标文件。

```
D:\>notepad myoutput.txt
```

3). 显示当前连接的所有Attached数据库和main数据库。

```
sqlite> ATTACH DATABASE 'D:/mydb.db' AS mydb;
sqlite> .databases
seq  name             file
---  ---------------  ------------------------
0    main
2    mydb              D:\mydb.db
```

4). 显示main数据库中的所有数据表。

```
sqlite> .tables
mytable
```

5). 显示匹配表名mytabl%的数据表的所有索引。

```
sqlite> CREATE INDEX myindex on mytable(first_col);
sqlite> .indices mytabl%
myindex
```

6). 显示匹配表名mytable%的数据表的Schema信息。

--依赖该表的索引信息也被输出。

```
sqlite> .schema mytabl%
CREATE TABLE mytable (first_col integer);
CREATE INDEX myindex on mytable(first_col);
```

7). 格式化显示SELECT的输出信息。

--插入测试数据

```
sqlite> INSERT INTO mytable VALUES(1);
sqlite> INSERT INTO mytable VALUES(2);
sqlite> INSERT INTO mytable VALUES(3);
```

--请注意没有任何设置时SELECT结果集的输出格式。

```
sqlite> SELECT * FROM mytable;
1
2
3
```

--显示SELECT结果集的列名。

--以列的形式显示各个字段。

--将其后输出的第一列显示宽度设置为10.

```
sqlite> .header on
sqlite> .mode column
sqlite> .width 10
sqlite> SELECT * FROM mytable;
first_col
----------
1
2
3
```

SQLite学习手册(在线备份)

一、常用备份：

下面的方法是比较简单且常用的SQLite数据库备份方式，见如下步骤：

1). 使用SQLite API或Shell工具在源数据库文件上加共享锁。

2). 使用Shell工具(cp或copy)拷贝数据库文件到备份目录。

3). 解除数据库文件上的共享锁。

以上3个步骤可以应用于大多数场景，而且速度也比较快，然而却存在一定的刚性缺陷，如：

1). 所有打算在源数据库上执行写操作的连接都不得不被挂起，直到整个拷贝过程结束并释放文件共享锁。

2). 不能拷贝数据到in-memory数据库。

3). 在拷贝过程中，一旦备份数据库所在的主机出现任何突发故障，备份数据库可能会被破坏。

在SQLite中提供了一组用于在线数据库备份的APIs函数(C接口)，可以很好的解决上述方法存在的不足。通过该组函数，可以将源数据库中的内容拷贝到另一个数据库，同时覆盖目标数据库中的数据。整个拷贝过程可以以增量的方式完成，在此情况下，源数据库也不需要在整个拷贝过程中都被加锁，而只是在真正读取数据时加共享锁。这样，其它的用户在访问源数据库时就不会被挂起。

二、在线备份APIs简介：

SQLite提供了以下3个APIs函数用于完成此操作，这里仅仅给出它们的基本用法，至于使用细节可以参考SQLite官方网站"APIs Reference"
(http://www.sqlite.org/c3ref/backup_finish.html)。

1). 函数sqlite3_backup_init()用于创建sqlite3_backup对象，该对象将作为本次拷贝操作的句柄传给其余两个函数。

2). 函数sqlite3_backup_step()用于数据拷贝，如果该函数的第二个参数为-1，那么整个拷贝过程都将在该函数的一次调用中完成。

3). 函数sqlite3_backup_finish()用于释放sqlite3_backup_init()函数申请的资源，以避免资源泄露。

在整个拷贝过程中如果出现任何错误，我们都可以通过调用目的数据库连接的sqlite3_errcode()函数来获取具体的错误码。此外，如果sqlite3_backup_step()调用失败，由于sqlite3_backup_finish()函数并不会修改当前连接的错误码，因此我们可以在调用

sqlite3_backup_finish()之后再获取错误码，从而在代码中减少了一次错误处理。见如下代码示例(来自SQLite官网)：

```
 1 /*
 2  This function is used to load the contents of a database file on disk
 3  into the "main" database of open database connection pInMemory, or
 4  to save the current contents of the database opened by pInMemory into
 5  a database file on disk. pInMemory is probably an in-memory database,
 6  but this function will also work fine if it is not.
 7 
 8  Parameter zFilename points to a nul-terminated string containing the
 9  name of the database file on disk to load from or save to. If parameter
10  isSave is non-zero, then the contents of the file zFilename are
11  overwritten with the contents of the database opened by pInMemory. If
12  parameter isSave is zero, then the contents of the database opened by
13  pInMemory are replaced by data loaded from the file zFilename.
14 
15  If the operation is successful, SQLITE_OK is returned. Otherwise, if
16  an error occurs, an SQLite error code is returned.
17 */
18 int loadOrSaveDb(sqlite3 *pInMemory, const char *zFilename, int isSave){
19   int rc;                   /* Function return code */
20   sqlite3 *pFile;           /* Database connection opened on zFilename */
21   sqlite3_backup *pBackup;  /* Backup object used to copy data */
22   sqlite3 *pTo;             /* Database to copy to (pFile or pInMemory) */
23   sqlite3 *pFrom;           /* Database to copy from (pFile or pInMemory) */
24 
25   /* Open the database file identified by zFilename. Exit early if this fails
26    for any reason. */
27   rc = sqlite3_open(zFilename, &pFile);
28   if( rc==SQLITE_OK ){
29 
30     /* If this is a 'load' operation (isSave==0), then data is copied
31      from the database file just opened to database pInMemory.
32      Otherwise, if this is a 'save' operation (isSave==1), then data
33      is copied from pInMemory to pFile.  Set the variables pFrom and
34      pTo accordingly. */
35     pFrom = (isSave ? pInMemory : pFile);
36     pTo   = (isSave ? pFile     : pInMemory);
37 
38     /* Set up the backup procedure to copy from the "main" database of
39      connection pFile to the main database of connection pInMemory.
40      If something goes wrong, pBackup will be set to NULL and an error
41      code and  message left in connection pTo.
42 
43      If the backup object is successfully created, call backup_step()
44      to copy data from pFile to pInMemory. Then call backup_finish()
45      to release resources associated with the pBackup object.  If an
46      error occurred, then  an error code and message will be left in
47      connection pTo. If no error occurred, then the error code belonging
48      to pTo is set to SQLITE_OK.
49 */
50     pBackup = sqlite3_backup_init(pTo, "main", pFrom, "main");
51     if( pBackup ){
52       (void)sqlite3_backup_step(pBackup, -1);
53       (void)sqlite3_backup_finish(pBackup);
54     }
55     rc = sqlite3_errcode(pTo);
56   }
57 
58   /* Close the database connection opened on database file zFilename
59    and return the result of this function. */
60   (void)sqlite3_close(pFile);
61   return rc;
62 }
```

三、高级应用技巧：

在上面的例子中，我们是通过sqlite3_backup_step()函数的一次调用完成了整个拷贝过程。该实现方式仍然存在之前说过的挂起其它写访问连接的问题，为了解决该问题，这里我们将继续介绍另外一种更高级的实现方式--分片拷贝，其实现步骤如下：

1). 函数sqlite3_backup_init()用于创建sqlite3_backup对象，该对象将作为本次拷贝操作的句柄传给其余两个函数。

2). 函数sqlite3_backup_step()被调用用于拷贝数据，和之前方法不同的是，该函数的第二个参数不再是-1，而是一个普通的正整数，表示每次调用将会拷贝的页面数量，如5。

3). 如果在函数sqlite3_backup_step()调用结束后，仍然有更多的页面需要被拷贝，那么我们将主动休眠250ms，然后再重复步骤2).

4). 函数sqlite3_backup_finish()用于释放sqlite3_backup_init()函数申请的资源，以避免资源泄露。

在上述步骤3)中我们主动休眠250ms，此期间，该拷贝操作不会在源数据库上持有任何读锁，这样其它的数据库连接在进行写操作时亦将不会被挂起。然而在休眠期间，如果另外一个线程或进程对源数据库进行了写操作，SQLite将会检测到该事件的发生，从而在下一次调用sqlite3_backup_step()函数时重新开始整个拷贝过程。唯一的例外是，如果源数据库不是in-memory数据库，同时写操作是在与拷贝操作同一个进程内完成，并且在操作时使用的也是同一个数据库连接句柄，那么目的数据库中数据也将被此操作同时自动修改。在下一次调用sqlite3_backup_step()时，也将不会有任何影响发生。

事实上，在SQLite中仍然提供了另外两个辅助性函数backup_remaining()和backup_pagecount()，其中前者将返回在当前备份操作中还有多少页面需要被拷贝，而后者将返回本次操作总共需要拷贝的页面数量。显而易见的是，通过这两个函数的返回结果，我们可以实时显示本次备份操作的整体进度，计算公式如下：

```
Completion = 100% * (pagecount() - remaining()) / pagecount()
```

见以下代码示例(来自SQLite官网)：

```
 1 /*
 2  Perform an online backup of database pDb to the database file named
 3  by zFilename. This function copies 5 database pages from pDb to
 4  zFilename, then unlocks pDb and sleeps for 250 ms, then repeats the
 5  process until the entire database is backed up.
 6
 7  The third argument passed to this function must be a pointer to a progress
 8  function. After each set of 5 pages is backed up, the progress function
 9  is invoked with two integer parameters: the number of pages left to
10  copy, and the total number of pages in the source file. This information
11  may be used, for example, to update a GUI progress bar.
12
13  While this function is running, another thread may use the database pDb, or
14  another process may access the underlying database file via a separate
15  connection.
16
17  If the backup process is successfully completed, SQLITE_OK is returned.
18  Otherwise, if an error occurs, an SQLite error code is returned.
19 */
20 int backupDb(
21   sqlite3 *pDb,               /* Database to back up */
22   const char *zFilename,      /* Name of file to back up to */
23   void(*xProgress)(int, int)  /* Progress function to invoke */
24 ){
25   int rc;                     /* Function return code */
26   sqlite3 *pFile;             /* Database connection opened on zFilename */
27   sqlite3_backup *pBackup;    /* Backup handle used to copy data */
28
29   /* Open the database file identified by zFilename. */
30   rc = sqlite3_open(zFilename, &pFile);
31   if( rc==SQLITE_OK ){
32
33     /* Open the sqlite3_backup object used to accomplish the transfer */
34     pBackup = sqlite3_backup_init(pFile, "main", pDb, "main");
35     if( pBackup ){
36
37       /* Each iteration of this loop copies 5 database pages from database
38        pDb to the backup database. If the return value of backup_step()
39        indicates that there are still further pages to copy, sleep for
40        250 ms before repeating. */
41       do {
42         rc = sqlite3_backup_step(pBackup, 5);
43         xProgress(
44             sqlite3_backup_remaining(pBackup),
45             sqlite3_backup_pagecount(pBackup)
46         );
47         if( rc==SQLITE_OK || rc==SQLITE_BUSY || rc==SQLITE_LOCKED ){
48           sqlite3_sleep(250);
49         }
50       } while( rc==SQLITE_OK || rc==SQLITE_BUSY || rc==SQLITE_LOCKED );
51
52       /* Release resources allocated by backup_init(). */
53       (void)sqlite3_backup_finish(pBackup);
54     }
55     rc = sqlite3_errcode(pFile);
56   }
57
58   /* Close the database connection opened on database file zFilename
59    and return the result of this function. */
60   (void)sqlite3_close(pFile);
61   return rc;
62 }
```

SQLite学习手册(内存数据库)

一、内存数据库：

在SQLite中，数据库通常是存储在磁盘文件中的。然而在有些情况下，我们可以让数据库始终驻留在内存中。最常用的一种方式是在调用sqlite3_open()的时候，数据库文件名参数传递":memory:"，如：

```
rc = sqlite3_open(":memory:", &db);
```

在调用完以上函数后，不会有任何磁盘文件被生成，取而代之的是，一个新的数据库在纯内存中被成功创建了。由于没有持久化，该数据库在当前数据库连接被关闭后就会立刻消失。需要注意的是，尽管多个数据库连接都可以通过上面的方法创建内存数据库，然而它们却是不同的数据库，相互之间没有任何关系。事实上，我们也可以通过Attach命令将内存数据库像其他普通数据库一样，附加到当前的连接中，如：

```
ATTACH DATABASE ':memory:' AS aux1;
```

二、临时数据库：

在调用sqlite3_open()函数或执行ATTACH命令时，如果数据库文件参数传的是空字符串，那么一个新的临时文件将被创建作为临时数据库的底层文件，如：

```
rc = sqlite3_open("", &db);
```

或

```
ATTACH DATABASE '' AS aux2;
```

和内存数据库非常相似，两个数据库连接创建的临时数据库也是各自独立的，在连接关闭后，临时数据库将自动消失，其底层文件也将被自动删除。

尽管磁盘文件被创建用于存储临时数据库中的数据信息，但是实际上临时数据库也会和内存数据库一样通常驻留在内存中，唯一不同的是，当临时数据库中数据量过大时，SQLite为了保证有更多的内存可用于其它操作，因此会将临时数据库中的部分数据写到磁盘文件中，而内存数据库则始终会将数据存放在内存中。

SQLite学习手册(临时文件)

一、简介：

尽管SQLite的数据库是由单一文件构成，然而事实上在SQLite运行时却存在着一些隐含的临时文件，这些临时文件是出于不同的目的而存在的，对于开发者而言，它们是透明的，因此在开发的过程中我们并不需要关注它们的存在。尽管如此，如果能对这些临时文件的产生机制和应用场景有着很好的理解，那么对我们今后应用程序的优化和维护都是极有帮助的。在SQLite中主要产生以下七种临时文件，如：

1). 回滚日志。

2). 主数据库日志。

3). SQL语句日志。

4). 临时数据库文件。

5). 视图和子查询的临时持久化文件。

6). 临时索引文件。

7). VACUUM命令使用的临时数据库文件。

二、具体说明：

1. 回滚日志：

SQLite为了保证事物的原子性提交和回滚，在事物开始时创建了该临时文件。此文件始终位于和数据库文件相同的目录下，其文件名格式为: 数据库文件名 + "-journal"。换句话说，如果没有该临时文件的存在，当程序运行的系统出现任何故障时，SQLite将无法保证事物的完整性，以及数据状态的一致性。该文件在事物提交或回滚后将被立刻删除。

在事物运行期间，如果当前主机因电源故障而宕机，而此时由于回滚日志文件已经保存在磁盘上，那么当下一次程序启动时，SQLite在打开数据库文件的过程中将会发现该临时文件的存在，我们称这种日志文件为"Hot Journal"。SQLite会在成功打开数据库之前先基于该文件完成数据库的恢复工作，以保证数据库的数据回复到上一个事物开始之前的状态。

在SQLite中我们可以通过修改journal_mode pragma，而使SQLite对维护该文件采用不同的策略。缺省情况下该值为DELETE，即在事物结束后删除日志文件。而PERSIST选项值将不会删除日志文件，而是将回滚日志文件的头部清零，从而避免了文件删除所带的磁盘开销。再有就是OFF选项值，该值将指示SQLite在开始事物时不产生回滚日志文件，这样一旦出现系统故障，SQLite也无法再保障数据库数据的一致性。

2. 主数据库日志：

在SQLite中，如果事物的操作作用于多个数据库，即通过ATTACH命令附加到当前连接中的数据库，那么SQLite将生成主数据库日志文件以保证事物产生的改变在多个数据库之间保持原子性。和回滚日志文件一样，主数据库日志文件也位于当前连接中主数据库文件所处的目录内，其文件名格式为：主数据库文件名 + 随机的后缀。在该文件中，将包含所有当前事物将会改变的Attached数据库的名字。在事物被提交之后，此文件亦被SQLite随之删除。

主数据库日志文件只有在某一事物同时操作多个数据库时(主数据库和Attached数据库)才有可能被创建。通过该文件，SQLite可以实现跨多个数据库的事物原子性，否则，只能简单的保证每个单一的数据库内的状态一致性。换句话说，如果该事物在执行的过程中出现系统崩溃或主机宕机的现象，在进行数据恢复时，若没有该文件的存在，将会导致部分SQLite数据库处于提交状态，而另外一部分则处于回滚状态，因此该事物的一致性将被打破。

3. SQL语句日志：

在一个较大的事物中，SQLite为了保证部分数据在出现错误时可以被正常回滚，所以在事物开始时创建了SQL语句日志文件。比如，update语句修改了前50条数据，然而在修改第51条数据时发现该操作将会破坏某字段的唯一性约束，最终SQLite将不得不通过该日志文件回滚已经修改的前50条数据。

SQL语句日志文件只有在INSERT或UPDATE语句修改多行记录时才有可能被创建，与此同时，这些操作还极有可能会打破某些约束并引发异常。但是如果INSERT或UPDATE语句没有被包含在BEGIN...COMMIT中，同时也没有任何其它的SQL语句正在当前的连接上运行，在这种情况下，SQLite将不会创建SQL语句日志文件，而是简单的通过回滚日志来完成部分数据的UNDO操作。

和上面两种临时文件不同的是，SQL语句日志文件并不一定要存储在和数据库文件相同的目录下，其文件名也是随机生成。该文件所占用的磁盘空间需要视UPDATE或INSERT语句将要修改的记录数量而定。在事物结束后，该文件将被自动删除。

4. 临时数据库文件：

当使用"CREATE TEMP TABLE"语法创建临时数据表时，该数据表仅在当前连接内可见，在当前连接被关闭后，临时表也随之消失。然而在生命期内，临时表将连同其相关的索引和视图均会被存储在一个临时的数据库文件之内。该临时文件是在第一次执行"CREATE TEMP TABLE"时即被创建的，在当前连接被关闭后，该文件亦将被自动删除。最后需要说明的是，临时数据库不能被执行DETACH命令，同时也不能被其它进程执行ATTACH命令。

5. 视图和子查询的临时持久化文件：

在很多包含子查询的查询中，SQLite的执行器会将该查询语句拆分为多个独立的SQL语句，同时将子查询的结果持久化到临时文件中，之后在基于该临时文件中的数据与外部查询进行关联，因此我们可以称其为物化子查询。通常而言，SQLite的优化器会尽力避免子查询的物化行为，但是在有些时候该操作是无法避免的。该临时文件所占用的磁盘空间需要依赖子查询检索出的数据数量，在查询结束后，该文件将被自动删除。见如下示例：

```
SELECT * FROM ex1 WHERE ex1.a IN (SELECT b FROM ex2);
```

在上面的查询语句中，子查询SELECT b FROM ex2的结果将会被持久化到临时文件中，外部查询在运行时将会为每一条记录去检查该临时文件，以判断当前记录是否出现在临时文件中，如果是则输出当前记录。显而易见的是，以上的行为将会产生大量的IO操作，从而显著的降低了查询的执行效率，为了避免临时文件的生成，我们可以将上面的查询语句改为：

```
SELECT * FROM ex1 WHERE EXISTS(SELECT 1 FROM ex2 WHERE ex2.b=ex1.a);
```

对于如下查询语句，如果SQLite不做任何智能的rewrite操作，该查询中的子查询也将会被持久化到临时文件中，如：

```
SELECT * FROM ex1 JOIN (SELECT b FROM ex2) AS t ON t.b=ex1.a;
```

在SQLite自动将其修改为下面的写法后，将不会再生成临时文件了，如：

```
SELECT ex1.*, ex2.b FROM ex1 JOIN ex2 ON ex2.b=ex1.a;
```

6. 临时索引文件：

当查询语句包含以下SQL从句时，SQLite为存储中间结果而创建了临时索引文件，如：

1). ORDER BY或GROUP BY从句。

2). 聚集查询中的DISTINCT关键字。

3). 由UNION、EXCEPT和INTERSECT连接的多个SELECT查询语句。

需要说明的是，如果在指定的字段上已经存在了索引，那么SQLite将不会再创建该临时索引文件，而是通过直接遍历索引来访问数据并提取有用信息。如果没有索引，则需要将排序的结果存储在临时索引文件中以供后用。该临时文件所占用的磁盘空间需要依赖排序数据的数量，在查询结束后，该文件被自动删除。

7. VACUUM命令使用的临时数据库文件：

VACUUM命令在工作时将会先创建一个临时文件，然后再将重建的整个数据库写入到该临时文件中。之后再将临时文件中的内容拷贝回原有的数据库文件中，最后删除该临时文件。

该临时文件所占用的磁盘空间不会超过原有文件的尺寸。

三、相关的编译时参数和指令：

对于SQLite来说，回滚日志、主数据库日志和SQL语句日志文件在需要的时候SQLite都会将它们写入磁盘文件，但是对于其它类型的临时文件，SQLite是可以将它们存放在内存中以取代磁盘文件的，这样在执行的过程中就可以减少大量的IO操作了。要完成该优化主要依赖于

以下三个因素：

1. 编译时参数SQLITE_TEMP_STORE：

该参数是源代码中的宏定义(#define)，其取值范围是0到3(缺省值为1)，见如下说明：

1). 等于0时，临时文件总是存储在磁盘上，而不会考虑temp_store pragma指令的设置。

2). 等于1时，临时文件缺省存储在磁盘上，但是该值可以被temp_store pragma指令覆盖。

3). 等于2时，临时文件缺省存储在内存中，但是该值可以被temp_store pragma指令覆盖。

4). 等于3时，临时文件总是存储在内存中，而不会考虑temp_store pragma指令的设置。

2. 运行时指令temp_store pragma：

该指令的取值范围是0到2(缺省值为0)，在程序运行时该指令可以被动态的设置，见如下说明：

1). 等于0时，临时文件的存储行为完全由SQLITE_TEMP_STORE编译期参数确定。

2). 等于1时，如果编译期参数SQLITE_TEMP_STORE指定使用内存存储临时文件，那么该指令将覆盖这一行为，使用磁盘存储。

2). 等于2时，如果编译期参数SQLITE_TEMP_STORE指定使用磁盘存储临时文件，那么该指令将覆盖这一行为，使用内存存储。

3. 临时文件的大小：

对于以上两个参数，都有参数值表示缺省情况是存储在内存中的，只有当临时文件的大小超过一定的阈值后才会根据一定的算法，将部分数据写入到磁盘中，以免临时文件占用过多的内存而影响其它程序的执行效率。

最后在重新赘述一遍，SQLITE_TEMP_STORE编译期参数和temp_store pragma运行时指令只会影响除回滚日志和主数据库日志之外的其它临时文件的存储策略。换句话说，回滚日志和主数据库日志将总是将数据写入磁盘，而不会关注以上两个参数的值。

四、其它优化策略：

在SQLite中由于采用了Page Cache的缓冲优化机制，因此即便临时文件被指定存储在磁盘上，也只有当该文件的大小增长到一定的尺寸后才有可能被SQLite刷新到磁盘文件上，在此之前它们仍将驻留在内存中。这就意味着对于大多数场景，如果临时表和临时索引的数据量相对较少，那么它们是不会被写到磁盘中的，当然也就不会有IO事件发生。只有当它们增长到内存不能容纳的时候才会被刷新到磁盘文件中的。其中SQLITE_DEFAULT_TEMP_CACHE_SIZE编译期参数可以用于指定临时表和索引在占用多少Cache Page时才需要被刷新到磁盘文件，该参数的缺省值为500页。

SQLite学习手册(锁和并发控制)

一、概述：

在SQLite中，锁和并发控制机制都是由pager_module模块负责处理的，如ACID(Atomic, Consistent, Isolated, and Durable)。在含有数据修改的事务中，该模块将确保或者所有的数据修改全部提交，或者全部回滚。与此同时，该模块还提供了一些磁盘文件的内存Cache功能。

事实上，pager_module模块并不关心数据库存储的细节，如B-Tree、编码方式、索引等，它只是将其视为由统一大小(通常为1024字节)的数据块构成的单一文件，其中每个块被称为一个页(page)。在该模块中页的起始编号为1，即第一个页的索引值是1，其后的页编号以此类推。

二、文件锁：

在SQLite的当前版本中，主要提供了以下五种方式的文件锁状态。

1). UNLOCKED：

文件没有持有任何锁，即当前数据库不存在任何读或写的操作。其它的进程可以在该数据库上执行任意的读写操作。此状态为缺省状态。

2). SHARED：

在此状态下，该数据库可以被读取但是不能被写入。在同一时刻可以有任意数量的进程在同一个数据库上持有共享锁，因此读操作是并发的。换句话说，只要有一个或多个共享锁处于活动状态，就不再允许有数据库文件写入的操作存在。

3). RESERVED：

假如某个进程在将来的某一时刻打算在当前的数据库中执行写操作，然而此时只是从数据库中读取数据，那么我们就可以简单的理解为数据库文件此时已经拥有了保留锁。当保留锁处于活动状态时，该数据库只能有一个或多个共享锁存在，即同一数据库的同一时刻只能存在一个保留锁和多个共享锁。在Oracle中此类锁被称之为预写锁，不同的是Oracle中锁的粒度可以细化到表甚至到行，因此该种锁在Oracle中对并发的影响程序不像SQLite中这样大。

4). PENDING：

PENDING锁的意思是说，某个进程正打算在该数据库上执行写操作，然而此时该数据库中却存在很多共享锁(读操作)，那么该写操作就必须处于等待状态，即等待所有共享锁消失为止，与此同时，新的读操作将不再被允许，以防止写锁饥饿的现象发生。在此等待期间，该数据库文件的锁状态为PENDING，在等到所有共享锁消失以后，PENDING锁状态的数据库文件将在获取排他锁之后进入EXCLUSIVE状态。

5). EXCLUSIVE：

在执行写操作之前，该进程必须先获取该数据库的排他锁。然而一旦拥有了排他锁，任何其它锁类型都不能与之共存。因此，为了最大化并发效率，SQLite将会最小化排他锁被持有的时间总量。

最后需要说明的是，和其它关系型数据库相比，如MySQL、Oracle等，SQLite数据库中所有的数据都存储在同一文件中，与此同时，它却仅仅提供了粗粒度的文件锁，因此，SQLite在并发性和伸缩性等方面和其它关系型数据库还是无法比拟的。由此可见，SQLite有其自身的适用场景，就如在本系列开篇中所说，它和其它关系型数据库之间的互换性还是非常有限的。

三、回滚日志：

当一个进程要改变数据库文件的时候，它首先将未改变之前的内容记录到回滚日志文件中。如果SQLite中的某一事务正在试图修改多个数据库中的数据，那么此时每一个数据库都将生成一个属于自己的回滚日志文件，用于分别记录属于自己的数据改变，与此同时还要生成一个用于协调多个数据库操作的主数据库日志文件，在主数据库日志文件中将包含各个数据库回滚日志文件的文件名，在每个回滚日志文件中也同样包含了主数据库日志文件的文件名信息。然而对于无需主数据库日志文件的回滚日志文件，其中也会保留主数据库日志文件的信息，只是此时该信息的值为空。

我们可以将回滚日志视为"HOT"日志文件，因为它的存在就是为了恢复数据库的一致性状态。当某一进程正在更新数据库时，应用程序或OS突然崩溃，这样更新操作就不能顺利完成。因此我们可以说"HOT"日志只有在异常条件下才会生成，如果一切都非常顺利的话，该文件将永远不会存在。

四、数据写入：

如果某一进程要想在数据库上执行写操作，那么必须先获取共享锁，在共享锁获取之后再获取保留锁。因为保留锁则预示着在将来某一时刻该进程将会执行写操作，所以在同一时刻只有一个进程可以持有一把保留锁，但是其它进程可以继续持有共享锁以完成数据读取的操作。如果要执行写操作的进程不能获取保留锁，那么这将说明另一进程已经获取了保留锁。在此种情况下，写操作将失败，并立即返回SQLITE_BUSY错误。在成功获取保留锁之后，该写进程将创建回滚日志。

在对任何数据作出改变之前，写进程会将待修改页中的原有内容先行写入回滚日志文件中，然而，这些数据发生变化的页起初并不会直接写入磁盘文件，而是保留在内存中，这样其它进程就可以继续读取该数据库中的数据了。

或者是因为内存中的cache已满，或者是应用程序已经提交了事务，最终，写进程将数据更新到数据库文件中。然而在此之前，写进程必须确保没有其它的进程正在读取数据库，同时回滚日志中的数据确实被物理的写入到磁盘文件中，其步骤如下：

1). 确保所有的回滚日志数据被物理的写入磁盘文件，以便在出现系统崩溃时可以将数据库恢复到一致的状态。

2). 获取PENDING锁，再获取排他锁，如果此时其它的进程仍然持有共享锁，写入线程将不得不被挂起并等待直到那些共享锁消失之后，才能进而得到排他锁。

3). 将内存中持有的修改页写出到原有的磁盘文件中。

如果写入到数据库文件的原因是因为cache已满，那么写入进程将不会立刻提交，而是继续对其它页进行修改。但是在接下来的修改被写入到数据库文件之前，回滚日志必须被再一次写到磁盘中。还要注意的是，写入进程获取到的排他锁必须被一直持有，直到所有的改变被提交时为止。这也意味着，从数据第一次被刷新到磁盘文件开始，直到事务被提交之前，其它的进程不能访问该数据库。

当写入进程准备提交时，将遵循以下步骤：

4). 获取排他锁，同时确保所有内存中的变化数据都被写入到磁盘文件中。

5). 将所有数据库文件的变化数据物理的写入到磁盘中。

6). 删除日志文件。如果在删除之前出现系统故障，进程在下一次打开该数据库时仍将基于该HOT日志进行恢复操作。因此只有在成功删除日志文件之后，我们才可以认为该事务成功完成。

7). 从数据库文件中删除所有的排他锁和PENDING锁。

一旦PENDING锁被释放，其它的进程就可以开始再次读取数据库了。

如果一个事务中包含多个数据库的修改，那么它的提交逻辑将更为复杂，见如下步骤：

4). 确保每个数据库文件都已经持有了排他锁和一个有效的日志文件。

5). 创建主数据库日志文件，同时将每个数据库的回滚日志文件的文件名写入到该主数据库日志文件中。

6). 再将主数据库日志文件的文件名分别写入到每个数据库回滚日志文件的指定位置中。

7). 将所有的数据库变化持久化到数据库磁盘文件中。

8). 删除主日志文件，如果在删除之前出现系统故障，进程在下一次打开该数据库时仍将基于该HOT日志进行恢复操作。因此只有在成功删除主日志文件之后，我们才可以认为该事务成功完成。

9). 删除每个数据库各自的日志文件。

10).从所有数据库中删除掉排他锁和PENDING锁。

最后需要说明的是，在SQLite2中，如果多个进程正在从数据库中读取数据，也就是说该数据库始终都有读操作发生，即在每一时刻该数据库都持有至少一把共享锁，这样将会导致没有任何进程可以执行写操作，因为在数据库持有读锁的时候是无法获取写锁的，我们将这种情形称为"写饥饿"。在SQLite3中，通过使用PENDING锁则有效的避免了"写饥饿"情形的发生。当某一进程持有PENDING锁时，已经存在的读操作可以继续进行，直到其正常结束，但是新的读操作将不会再被SQLite接受，所以在已有的读操作全部结束后，持有PENDING锁的进程就可以被激活并试图进一步获取排他锁以完成数据的修改操作。

五、SQL级别的事务控制：

SQLite3在实现上确实针对锁和并发控制做出了一些精巧的变化，特别是对于事务这一SQL语言级别的特征。在缺省情况下，SQLite3会将所有的SQL操作置于antocommit模式下，这样所有针对数据库的修改操作都会在SQL命令执行结束后被自动提交。在SQLite中，SQL命令"BEGIN TRANSACTION"用于显式的声明一个事务，即其后的SQL语句在执行后都不会自动提交，而是需要等到SQL命令"COMMIT"或"ROLLBACK"被执行时，才考虑提交还是回滚。由此可以推断出，在BEGIN命令被执行后并没有立即获得任何类型的锁，而是在执行第一个SELECT语句时才得到一个共享锁，或者是在执行第一个DML语句时才获得一个保留锁。至于排它锁，只有在数据从内存写入磁盘时开始，直到事务提交或回滚之前才能持有排它锁。

如果多个SQL命令在同一个时刻同一个数据库连接中被执行，autocommit将会被延迟执行，直到最后一个命令完成。比如，如果一个SELECT语句正在被执行，在这个命令执行期间，需要返回所有检索出来的行记录，如果此时处理结果集的线程因为业务逻辑的需要被暂时挂起并处于等待状态，而其它的线程此时或许正在该连接上对该数据库执行INSERT、UPDATE或DELETE命令，那么所有这些命令作出的数据修改都必须等到SELECT检索结束后才能被提交。

这是该系列中最后一篇关于*SQLite*理论与应用方面的博客了，后面将发布两篇关于如何利用*SQLite*编程的博客，其中还将包含四个典型的应用代码示例，望大家继续保持关注。

SQLite学习手册(实例代码<一>)

一、获取表的Schema信息：

1). 动态创建表。

2). 根据sqlite3提供的API，获取表字段的信息，如字段数量以及每个字段的类型。

3). 删除该表。

见以下代码及关键性注释：

```
 1 #include <sqlite3.h>
 2 #include <string>
 3
 4 using namespace std;
 5
 6 void doTest()
 7 {
 8     sqlite3* conn = NULL;
 9     //1\. 打开数据库
10     int result = sqlite3_open("D:/mytest.db",&conn);
11     if (result != SQLITE_OK) {
12         sqlite3_close(conn);
13         return;
14     }
15     const char* createTableSQL =
16         "CREATE TABLE TESTTABLE (int_col INT, float_col REAL, string_col TEXT)";
17     sqlite3_stmt* stmt = NULL;
18     int len = strlen(createTableSQL);
19     //2\. 准备创建数据表，如果创建失败，需要用sqlite3_finalize释放sqlite3_stmt对象，以防止内存泄
20     if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
21         if (stmt)
22             sqlite3_finalize(stmt);
23         sqlite3_close(conn);
24         return;
25     }
26     //3\. 通过sqlite3_step命令执行创建表的语句。对于DDL和DML语句而言，sqlite3_step执行正确的返回
27     //只有SQLITE_DONE，对于SELECT查询而言，如果有数据返回SQLITE_ROW，当到达结果集末尾时则返回
28     //SQLITE_DONE。
29     if (sqlite3_step(stmt) != SQLITE_DONE) {
30         sqlite3_finalize(stmt);
31         sqlite3_close(conn);
32         return;
33     }
34     //4\. 释放创建表语句对象的资源。
35     sqlite3_finalize(stmt);
36     printf("Succeed to create test table now.\n");
37     //5\. 构造查询表数据的sqlite3_stmt对象。
38     const char* selectSQL = "SELECT * FROM TESTTABLE WHERE 1 = 0";
39     sqlite3_stmt* stmt2 = NULL;
40     if (sqlite3_prepare_v2(conn,selectSQL,strlen(selectSQL),&stmt2,NULL) != SQLITE_OK)
41         if (stmt2)
42             sqlite3_finalize(stmt2);
43         sqlite3_close(conn);
44         return;
45     }
46     //6\. 根据select语句的对象，获取结果集中的字段数量。
47     int fieldCount = sqlite3_column_count(stmt2);
48     printf("The column count is %d.\n",fieldCount);
49     //7\. 遍历结果集中每个字段meta信息，并获取其声明时的类型。
50     for (int i = 0; i < fieldCount; ++i) {
51         //由于此时Table中并不存在数据，再有就是SQLite中的数据类型本身是动态的，所以在没有数据时
```

```
52          //无法通过sqlite3_column_type函数获取，此时sqlite3_column_type只会返回SQLITE_NULL，
53          //直到有数据时才能返回具体的类型，因此这里使用了sqlite3_column_decltype函数来获取表声
54          //明时给出的声明类型。
55          string stype = sqlite3_column_decltype(stmt2,i);
56          stype = strlwr((char*)stype.c_str());
57          //下面的解析规则见该系列的“数据类型-->1\. 决定字段亲缘性的规则”部分，其链接如下：
58          //http://www.cnblogs.com/stephen-liu74/archive/2012/01/18/2325258.html
59          if (stype.find("int") != string::npos) {
60              printf("The type of %dth column is INTEGER.\n",i);
61          } else if (stype.find("char") != string::npos
62              || stype.find("text") != string::npos) {
63              printf("The type of %dth column is TEXT.\n",i);
64          } else if (stype.find("real") != string::npos
65              || stype.find("floa") != string::npos
66              || stype.find("doub") != string::npos ) {
67              printf("The type of %dth column is DOUBLE.\n",i);
68          }
69      }
70      sqlite3_finalize(stmt2);
71      //8\. 为了方便下一次测试运行，我们这里需要删除该函数创建的数据表，否则在下次运行时将无法
72      //创建该表，因为它已经存在。
73      const char* dropSQL = "DROP TABLE TESTTABLE";
74      sqlite3_stmt* stmt3 = NULL;
75      if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt3,NULL) != SQLITE_OK) {
76          if (stmt3)
77              sqlite3_finalize(stmt3);
78          sqlite3_close(conn);
79          return;
80      }
81      if (sqlite3_step(stmt3) == SQLITE_DONE) {
82          printf("The test table has been dropped.\n");
83      }
84      sqlite3_finalize(stmt3);
85      sqlite3_close(conn);
86 }
87
88 int main()
89 {
90     doTest();
91     return 0;
92 }
93 //输出结果为：
94 //Succeed to create test table now.
95 //The column count is 3.
96 //The type of 0th column is INTEGER.
97 //The type of 1th column is DOUBLE.
98 //The type of 2th column is TEXT.
99 //The test table has been dropped.
```

二、常规数据插入：

1). 创建测试数据表。

2). 通过INSERT语句插入测试数据。

3). 删除测试表。

见以下代码及关键性注释：

```
1 #include <sqlite3.h>
2 #include <string>
3 #include <stdio.h>
4
5 using namespace std;
6
```

```
 7 void doTest()
 8 {
 9     sqlite3* conn = NULL;
10     //1\. 打开数据库
11     int result = sqlite3_open("D:/mytest.db",&conn);
12     if (result != SQLITE_OK) {
13         sqlite3_close(conn);
14         return;
15     }
16     const char* createTableSQL =
17         "CREATE TABLE TESTTABLE (int_col INT, float_col REAL, string_col TEXT)";
18     sqlite3_stmt* stmt = NULL;
19     int len = strlen(createTableSQL);
20     //2\. 准备创建数据表，如果创建失败，需要用sqlite3_finalize释放sqlite3_stmt对象，以防止内存泄
21     if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
22         if (stmt)
23             sqlite3_finalize(stmt);
24         sqlite3_close(conn);
25         return;
26     }
27     //3\. 通过sqlite3_step命令执行创建表的语句。对于DDL和DML语句而言，sqlite3_step执行正确的返回
28     //只有SQLITE_DONE，对于SELECT查询而言，如果有数据返回SQLITE_ROW，当到达结果集末尾时则返回
29     //SQLITE_DONE。
30     if (sqlite3_step(stmt) != SQLITE_DONE) {
31         sqlite3_finalize(stmt);
32         sqlite3_close(conn);
33         return;
34     }
35     //4\. 释放创建表语句对象的资源。
36     sqlite3_finalize(stmt);
37     printf("Succeed to create test table now.\n");
38 
39     int insertCount = 10;
40     //5\. 构建插入数据的sqlite3_stmt对象。
41     const char* insertSQL = "INSERT INTO TESTTABLE VALUES(%d,%f,'%s')";
42     const char* testString = "this is a test.";
43     char sql[1024];
44     sqlite3_stmt* stmt2 = NULL;
45     for (int i = 0; i < insertCount; ++i) {
46         sprintf(sql,insertSQL,i,i * 1.0,testString);
47         if (sqlite3_prepare_v2(conn,sql,strlen(sql),&stmt2,NULL) != SQLITE_OK) {
48             if (stmt2)
49                 sqlite3_finalize(stmt2);
50             sqlite3_close(conn);
51             return;
52         }
53         if (sqlite3_step(stmt2) != SQLITE_DONE) {
54             sqlite3_finalize(stmt2);
55             sqlite3_close(conn);
56             return;
57         }
58         printf("Insert Succeed.\n");
59     }
60     sqlite3_finalize(stmt2);
61     //6\. 为了方便下一次测试运行，我们这里需要删除该函数创建的数据表，否则在下次运行时将无法
62     //创建该表，因为它已经存在。
63     const char* dropSQL = "DROP TABLE TESTTABLE";
64     sqlite3_stmt* stmt3 = NULL;
65     if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt3,NULL) != SQLITE_OK) {
66         if (stmt3)
67             sqlite3_finalize(stmt3);
68         sqlite3_close(conn);
69         return;
70     }
71     if (sqlite3_step(stmt3) == SQLITE_DONE) {
72         printf("The test table has been dropped.\n");
73     }
74     sqlite3_finalize(stmt3);
75     sqlite3_close(conn);
76 }
77 
78 int main()
```

```
79 {
80     doTest();
81     return 0;
82 }
83 //输出结果如下：
84 //Succeed to create test table now.
85 //Insert Succeed.
86 //Insert Succeed.
87 //Insert Succeed.
88 //Insert Succeed.
89 //Insert Succeed.
90 //Insert Succeed.
91 //Insert Succeed.
92 //Insert Succeed.
93 //Insert Succeed.
94 //Insert Succeed.
95 //The test table has been dropped.
```

SQLite学习手册(实例代码<二>)

三、高效的批量数据插入：

在给出操作步骤之前先简单说明一下批量插入的概念，以帮助大家阅读其后的示例代码。事实上，批量插入并不是什么新的概念，在其它关系型数据库的C接口API中都提供了一定的支持，只是接口的实现方式不同而已。纵观众多流行的数据库接口，如OCI(Oracle API)、MySQL API和PostgreSQL API等，OCI提供的编程接口最为方便，实现方式也最为高效。SQLite作为一种简单灵活的嵌入式数据库也同样提供了该功能，但是实现方式并不像其他数据库那样方便明显，它只是通过一种隐含的技巧来达到批量插入的目的，其逻辑如下：

1). 开始一个事物，以保证后面的数据操作语句均在该事物内完成。在SQLite中，如果没有手工开启一个事物，其所有的DML语句都是在自动提交模式下工作的，既每次操作后数据均被自动提交并写入磁盘文件。然而在非自动提交模式下，只有当其所在的事物被手工COMMIT之后才会将修改的数据写入到磁盘中，之前修改的数据都是仅仅驻留在内存中。显而易见，这样的批量写入方式在效率上势必会远远优于多迭代式的单次写入操作。

2). 基于变量绑定的方式准备待插入的数据，这样可以节省大量的sqlite3_prepare_v2函数调用次数，从而节省了多次将同一SQL语句编译成SQLite内部识别的字节码所用的时间。事实上，SQLite的官方文档中已经明确指出，在很多时候sqlite3_prepare_v2函数的执行时间要多于sqlite3_step函数的执行时间，因此建议使用者要尽量避免重复调用sqlite3_prepare_v2函数。在我们的实现中，如果想避免此类开销，只需将待插入的数据以变量的形式绑定到SQL语句中，这样该SQL语句仅需调用sqlite3_prepare_v2函数编译一次即可，其后的操作只是替换不同的变量数值。

3). 在完成所有的数据插入后显式的提交事物。提交后，SQLite会将当前连接自动恢复为自动提交模式。

下面是示例代码的实现步骤：

1). 创建测试数据表。

2). 通过执行BEGIN TRANSACTION语句手工开启一个事物。

3). 准备插入语句及相关的绑定变量。

4). 迭代式插入数据。

5). 完成后通过执行COMMIT语句提交事物。

6). 删除测试表。

见以下代码及关键性注释：

```
1 #include <sqlite3.h>
```

```
2 #include <string>
3 #include <stdio.h>
4
5 using namespace std;
6
7 void doTest()
8 {
9     sqlite3* conn = NULL;
10    //1\. 打开数据库
11    int result = sqlite3_open("D:/mytest.db",&conn);
12    if (result != SQLITE_OK) {
13        sqlite3_close(conn);
14        return;
15    }
16    const char* createTableSQL =
17        "CREATE TABLE TESTTABLE (int_col INT, float_col REAL, string_col TEXT)";
18    sqlite3_stmt* stmt = NULL;
19    int len = strlen(createTableSQL);
20    //2\. 准备创建数据表，如果创建失败，需要用sqlite3_finalize释放sqlite3_stmt对象，以防止内存
21    if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
22        if (stmt)
23            sqlite3_finalize(stmt);
24        sqlite3_close(conn);
25        return;
26    }
27    //3\. 通过sqlite3_step命令执行创建表的语句。对于DDL和DML语句而言，sqlite3_step执行正确的返
28    //只有SQLITE_DONE，对于SELECT查询而言，如果有数据返回SQLITE_ROW，当到达结果集末尾时则返回
29    //SQLITE_DONE。
30    if (sqlite3_step(stmt) != SQLITE_DONE) {
31        sqlite3_finalize(stmt);
32        sqlite3_close(conn);
33        return;
34    }
35    //4\. 释放创建表语句对象的资源。
36    sqlite3_finalize(stmt);
37    printf("Succeed to create test table now.\n");
38
39    //5\. 显式的开启一个事物。
40    sqlite3_stmt* stmt2 = NULL;
41    const char* beginSQL = "BEGIN TRANSACTION";
42    if (sqlite3_prepare_v2(conn,beginSQL,strlen(beginSQL),&stmt2,NULL) != SQLITE_OK)
43        if (stmt2)
44            sqlite3_finalize(stmt2);
45        sqlite3_close(conn);
46        return;
47    }
48    if (sqlite3_step(stmt2) != SQLITE_DONE) {
49        sqlite3_finalize(stmt2);
50        sqlite3_close(conn);
51        return;
52    }
53    sqlite3_finalize(stmt2);
54
55    //6\. 构建基于绑定变量的插入数据。
56    const char* insertSQL = "INSERT INTO TESTTABLE VALUES(?,?,?)";
57    sqlite3_stmt* stmt3 = NULL;
58    if (sqlite3_prepare_v2(conn,insertSQL,strlen(insertSQL),&stmt3,NULL) != SQLITE_OK
59        if (stmt3)
60            sqlite3_finalize(stmt3);
61        sqlite3_close(conn);
62        return;
63    }
64    int insertCount = 10;
65    const char* strData = "This is a test.";
66    //7\. 基于已有的SQL语句，迭代的绑定不同的变量数据
67    for (int i = 0; i < insertCount; ++i) {
68        //在绑定时，最左面的变量索引值是1。
69        sqlite3_bind_int(stmt3,1,i);
70        sqlite3_bind_double(stmt3,2,i * 1.0);
71        sqlite3_bind_text(stmt3,3,strData,strlen(strData),SQLITE_TRANSIENT);
72        if (sqlite3_step(stmt3) != SQLITE_DONE) {
73            sqlite3_finalize(stmt3);
```

```
 74             sqlite3_close(conn);
 75             return;
 76         }
 77         //重新初始化该sqlite3_stmt对象绑定的变量。
 78         sqlite3_reset(stmt3);
 79         printf("Insert Succeed.\n");
 80     }
 81     sqlite3_finalize(stmt3);
 82 
 83     //8\. 提交之前的事物。
 84     const char* commitSQL = "COMMIT";
 85     sqlite3_stmt* stmt4 = NULL;
 86     if (sqlite3_prepare_v2(conn,commitSQL,strlen(commitSQL),&stmt4,NULL) != SQLITE_OK
 87         if (stmt4)
 88             sqlite3_finalize(stmt4);
 89         sqlite3_close(conn);
 90         return;
 91     }
 92     if (sqlite3_step(stmt4) != SQLITE_DONE) {
 93         sqlite3_finalize(stmt4);
 94         sqlite3_close(conn);
 95         return;
 96     }
 97     sqlite3_finalize(stmt4);
 98 
 99     //9\. 为了方便下一次测试运行，我们这里需要删除该函数创建的数据表，否则在下次运行时将无法
100     //创建该表，因为它已经存在。
101     const char* dropSQL = "DROP TABLE TESTTABLE";
102     sqlite3_stmt* stmt5 = NULL;
103     if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt5,NULL) != SQLITE_OK) {
104         if (stmt5)
105             sqlite3_finalize(stmt5);
106         sqlite3_close(conn);
107         return;
108     }
109     if (sqlite3_step(stmt5) == SQLITE_DONE) {
110         printf("The test table has been dropped.\n");
111     }
112     sqlite3_finalize(stmt5);
113     sqlite3_close(conn);
114 }
115 
116 int main()
117 {
118     doTest();
119     return 0;
120 }
121 //输出结果如下：
122 //Succeed to create test table now.
123 //Insert Succeed.
124 //Insert Succeed.
125 //Insert Succeed.
126 //Insert Succeed.
127 //Insert Succeed.
128 //Insert Succeed.
129 //Insert Succeed.
130 //Insert Succeed.
131 //Insert Succeed.
132 //Insert Succeed.
133 //The test table has been dropped.
```

该结果和上一个例子(普通数据插入)的结果完全相同，只是在执行效率上明显优于前者。

四、数据查询：

数据查询是每个关系型数据库都会提供的最基本功能，下面的代码示例将给出如何通过SQLite API获取数据。

1). 创建测试数据表。

2). 插入一条测试数据到该数据表以便于后面的查询。

3). 执行SELECT语句检索数据。

4). 删除测试表。

见以下示例代码和关键性注释：

```
 1 #include <sqlite3.h>
 2 #include <string>
 3 #include <stdio.h>
 4 
 5 using namespace std;
 6 
 7 void doTest()
 8 {
 9     sqlite3* conn = NULL;
10     //1\. 打开数据库
11     int result = sqlite3_open("D:/mytest.db",&conn);
12     if (result != SQLITE_OK) {
13         sqlite3_close(conn);
14         return;
15     }
16     const char* createTableSQL =
17         "CREATE TABLE TESTTABLE (int_col INT, float_col REAL, string_col TEXT)";
18     sqlite3_stmt* stmt = NULL;
19     int len = strlen(createTableSQL);
20     //2\. 准备创建数据表，如果创建失败，需要用sqlite3_finalize释放sqlite3_stmt对象，以防止内存
21     if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
22         if (stmt)
23             sqlite3_finalize(stmt);
24         sqlite3_close(conn);
25         return;
26     }
27     //3\. 通过sqlite3_step命令执行创建表的语句。对于DDL和DML语句而言，sqlite3_step执行正确的返
28     //只有SQLITE_DONE，对于SELECT查询而言，如果有数据返回SQLITE_ROW，当到达结果集末尾时则返回
29     //SQLITE_DONE。
30     if (sqlite3_step(stmt) != SQLITE_DONE) {
31         sqlite3_finalize(stmt);
32         sqlite3_close(conn);
33         return;
34     }
35     //4\. 释放创建表语句对象的资源。
36     sqlite3_finalize(stmt);
37     printf("Succeed to create test table now.\n");
38 
39     //5\. 为后面的查询操作插入测试数据。
40     sqlite3_stmt* stmt2 = NULL;
41     const char* insertSQL = "INSERT INTO TESTTABLE VALUES(20,21.0,'this is a test.')"
42     if (sqlite3_prepare_v2(conn,insertSQL,strlen(insertSQL),&stmt2,NULL) != SQLITE_OK
43         if (stmt2)
44             sqlite3_finalize(stmt2);
45         sqlite3_close(conn);
46         return;
47     }
48     if (sqlite3_step(stmt2) != SQLITE_DONE) {
49         sqlite3_finalize(stmt2);
50         sqlite3_close(conn);
51         return;
52     }
53     printf("Succeed to insert test data.\n");
54     sqlite3_finalize(stmt2);
55 
56     //6\. 执行SELECT语句查询数据。
57     const char* selectSQL = "SELECT * FROM TESTTABLE";
58     sqlite3_stmt* stmt3 = NULL;
```

```
 59     if (sqlite3_prepare_v2(conn,selectSQL,strlen(selectSQL),&stmt3,NULL) != SQLITE_OK
 60         if (stmt3)
 61             sqlite3_finalize(stmt3);
 62         sqlite3_close(conn);
 63         return;
 64     }
 65     int fieldCount = sqlite3_column_count(stmt3);
 66     do {
 67         int r = sqlite3_step(stmt3);
 68         if (r == SQLITE_ROW) {
 69             for (int i = 0; i < fieldCount; ++i) {
 70                 //这里需要先判断当前记录当前字段的类型，再根据返回的类型使用不同的API函数
 71                 //获取实际的数据值。
 72                 int vtype = sqlite3_column_type(stmt3,i);
 73                 if (vtype == SQLITE_INTEGER) {
 74                     int v = sqlite3_column_int(stmt3,i);
 75                     printf("The INTEGER value is %d.\n",v);
 76                 } else if (vtype == SQLITE_FLOAT) {
 77                     double v = sqlite3_column_double(stmt3,i);
 78                     printf("The DOUBLE value is %f.\n",v);
 79                 } else if (vtype == SQLITE_TEXT) {
 80                     const char* v = (const char*)sqlite3_column_text(stmt3,i);
 81                     printf("The TEXT value is %s.\n",v);
 82                 } else if (vtype == SQLITE_NULL) {
 83                     printf("This value is NULL.\n");
 84                 }
 85             }
 86         } else if (r == SQLITE_DONE) {
 87             printf("Select Finished.\n");
 88             break;
 89         } else {
 90             printf("Failed to SELECT.\n");
 91             sqlite3_finalize(stmt3);
 92             sqlite3_close(conn);
 93             return;
 94         }
 95     } while (true);
 96     sqlite3_finalize(stmt3);
 97 
 98     //7\. 为了方便下一次测试运行，我们这里需要删除该函数创建的数据表，否则在下次运行时将无法
 99     //创建该表，因为它已经存在。
100     const char* dropSQL = "DROP TABLE TESTTABLE";
101     sqlite3_stmt* stmt4 = NULL;
102     if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt4,NULL) != SQLITE_OK) {
103         if (stmt4)
104             sqlite3_finalize(stmt4);
105         sqlite3_close(conn);
106         return;
107     }
108     if (sqlite3_step(stmt4) == SQLITE_DONE) {
109         printf("The test table has been dropped.\n");
110     }
111     sqlite3_finalize(stmt4);
112     sqlite3_close(conn);
113 }
114 
115 int main()
116 {
117     doTest();
118     return 0;
119 }
120 //输出结果如下：
121 //Succeed to create test table now.
122 //Succeed to insert test data.
123 //The INTEGER value is 20.
124 //The DOUBLE value is 21.000000.
125 //The TEXT value is this is a test..
126 //Select Finished.
127 //The test table has been dropped.
```